Modal Logics and Types: Looking Back and Looking Forward

Frank Pfenning

Computer Science Department
Carnegie Mellon University

PEPM'22
January 17, 2022
Invited Talk

Outline

- S4 and Runtime Code Generation
- Decomposing Modalities
- Lax Logic and Monadic Programming
- Adjoint Logic
- K and Partial Evaluation
- Substructural Adjoint Logic

Compiling the Polymorphic λ-Calculus

Spiro Michaylov and Frank Pfenning

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, U.S.A.

Internet: spiro@cs.cmu.edu and fp@cs.cmu.edu

Abstract

We report some initial results regarding the efficient compilation of the second-order polymorphic λ-calculus (F_2). Our compiler makes strong use of type information and the strong normalization and Church-Rosser properties of F_2. Among the conceptual tools we develop is a notion of observational equivalence for F_2, which we use to outline a proof that our compiler preserves the observable behavior of programs. Our technique compiles functions of well-understood inductive types to non-functional data structures, and computation is no longer just β-reduction. A limited form of partial evaluation with a simple "Eureka" step is used to help circumvent provable inefficiencies of some functions in the pure polymorphic λ-calculus. For example, the usual predecessor function on Church numerals is compiled to a constant-time function.

1 Introduction

quantification, side-effects, and assignment. These features destroy many of its elegant and useful properties, and it may be more accurate to say that such languages are "inspired by" rather than "based on" the polymorphic λ-calculus. The extensions are justified by the goal of designing and implementing a practical and efficient language, two criteria which, conventional wisdom holds, are clearly not satisfied by the polymorphic λ-calculus. This general impression about inefficiency has been formalized and proved by Parigot [14] who shows that there is no constant time definition of the predecessor function over Church's representation of natural numbers. Attempts at overcoming this inefficiency have been unsatisfactory either because other functions become inefficient, they give up typing, or they require built-in inductive data type definition and primitive recursion (see, for example, [14] or [6]).

In this paper we describe the principles behind a compiler for the pure second-order polymorphic λ-calculus (F_2). The original motivation for developing this compiler was to execute proofs extracted from proofs in the Calculus of Constructions, although we consider the techniques we have developed to be of

Lightweight Run-Time Code Generation*

Mark Leone Peter Lee

Carnegie Mellon University
Pittsburgh, Pennsylvania 15213 USA
{mleone,petel}@cs.cmu.edu

Abstract

Run-time code generation is an alternative and complement to compile-time program analysis and optimization. Static analyses are inherently imprecise because most interesting aspects of run-time behavior are uncomputable. By deferring aspects of compilation to run time, more precise information about program behavior can be exploited, leading to greater opportunities for code improvement.

The cost of performing optimization at run time is of paramount importance, since it must be repaid by improved performance in order to obtain an overall speedup. This paper describes a lightweight approach to run-time code generation, called *deferred compilation*, in which compile-time specialization is employed to reduce the cost of optimizing and generating code at run time. Implementation strategies developed for a prototype compiler are discussed, and the results of preliminary experiments demonstrating significant overall speedup are presented.

1 Introduction

Many compiler optimizations depend on compile-time analysis to approximate properties of a program's run-time behavior. Static analyses are necessarily imprecise because most

In this paper we report on our experience with a new approach to generating optimized code at run time. The salient characteristics of our approach, which we term *deferred compilation*, are as follows:

- It is *lightweight*. Compile-time specialization eliminates the need to process any intermediate representation of a program at run time. Each part of a compiled program that performs run-time code generation is "hard wired" to optimize and generate code for a small portion of the input program.
- It is largely *automatic*. Manual construction of code templates or run-time code generators is not required. Syntactic cues and programmer hints are used to determine which parts of a program should be subjected to run-time compilation.
- It is *general*. Many standard optimizations, such as strength reduction and function inlining, can be efficiently employed at run time.

We have implemented a prototype compiler, which we call FABIUS, to evaluate this approach. In preliminary experiments, we have found that the overhead of deferred compilation is often quite small when compared to the per-

Runtime Code Generation

- Mark Leone and Peter Lee: *Lightweight Runtime Code Generation*, PEPM 1994.
  - Deferred compilation in Fabius compiler for ML
  - Staging analysis similar to binding-time analysis in partial evaluation
  - Example: from matrix multiplication to sparse matrix multiplication
  - Example: the power function (used in this talk)
- Privately: Sometimes things are faster, sometimes slower. Can we express and enforce proper staging in the type system?

On the Logical Foundations of Staged Computation

Frank Pfenning

PEPM'00, Boston, MA
January 22, 2000

1. Introduction
2. Judgments and Propositions
3. Intensional Types
4. Run-Time Code Generation
5. The PML Compiler
6. Conclusion

1

Curried Functions

- Express some staging but do not enforce
- A function $\tau \to (\sigma \to \rho)$ takes a value of type τ and returns a code generator of type $\sigma \to \rho$.
- The *power* function, first version

  $$\mathit{power} : \mathsf{nat} \to (\mathsf{nat} \to \mathsf{nat})$$
  $$\mathit{power} = \lambda b.\, \lambda e.\, \mathbf{if}\ e = 0\ \mathbf{then}\ 1\ \mathbf{else}\ b * \mathit{power}\ b\ (e - 1)$$

  $$\mathit{power}\ 2 \mapsto^* \lambda e.\, \mathbf{if}\ e = 0\ \mathbf{then}\ 1\ \mathbf{else}\ 2 * \mathit{power}\ 2\ (e - 1)$$

- Possibly better: swap arguments

  $$\mathit{power} : \mathsf{nat} \to (\mathsf{nat} \to \mathsf{nat})$$
  $$\mathit{power} = \lambda e.\, \lambda b.\, \mathbf{if}\ e = 0\ \mathbf{then}\ 1\ \mathbf{else}\ b * \mathit{power}\ b\ (e - 1)$$

  $$\mathit{power}\ 2 \mapsto^* \lambda b.\, \mathbf{if}\ 2 = 0\ \mathbf{then}\ 1\ \mathbf{else}\ b * \mathit{power}\ b\ (2 - 1)$$

Restaging

- Restage

$$
\begin{aligned}
&\mathit{power} : \mathsf{nat} \to (\mathsf{nat} \to \mathsf{nat}) \\
&\mathit{power} = \lambda e.\ \ \textbf{if } e = 0 \textbf{ then } \lambda b.\, 1 \\
&\qquad\qquad\qquad \textbf{else let } f = \mathit{power}\ (e - 1) \textbf{ in } \lambda b.\, b * f\ b
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{power}\ 0 \mapsto^* \lambda b_0.\, 1 \\
&\mathit{power}\ 1 \mapsto^* \lambda b_1.\, b_1 * (\lambda b_0.\, 1)\ b_1 \\
&\mathit{power}\ 2 \mapsto^* \lambda b_2.\, b_2 * (\lambda b_1.\, b_1 * (\lambda b_0.\, 1)\ b_1)\ b_2 \\
&\qquad\quad =_{\beta_v} \lambda b_2.\, b_2 * b_2 * 1
\end{aligned}
$$

- If we could ensure we get the source of the result, we may be able to generate pretty good code

Quotation

- Frank Pfenning. *On the Logical Foundations of Staged Computation*, PEPM 2000. [Davies & Pf 1996]
- Insight: In order to compile at runtime we must have a quoted source expression
- Quotation is analyzed in (classical) modal logic
- Are there suitable intuitionistic versions?
- If so, which of the many modal logics should it be?

Example Revisited

- Type $\Box\tau$ for a (source) expression of type τ

  $power$: nat $\to \Box$(nat $\to$ nat)
  (code later)

- Axiomatically

$$\frac{\vdash A}{\vdash \Box A}\ \text{Nec}$$

$\vdash \Box(A \to B) \to (\Box A \to \Box B)$ (dist$\to$)
$\vdash \Box A \to A$ (eval, axiom T)
$\vdash \Box A \to \Box\Box A$ (requote, axiom 4)

$\nvdash A \to \Box A$ (but true for some A)

- An intuitionistic version of S4!

A Curry-Howard Correspondence

- Prefer a system of natural deduction
- Add term assignment and operational semantics
  [Pf & Wong 1995] [Ghani, de Paiva, Ritter 1998] [Davies & Pf 2001]
- Erasing terms yields a system of natural deduction for S4
- Principal judgment

$$\underbrace{u_1 :: \sigma_1, \ldots, u_m :: \sigma_m}_{\Delta} \;;\; \underbrace{x_1 : \tau_1, \ldots, x_n : \tau_n}_{\Gamma} \vdash e : \tau$$

- u_j will be bound to source expressions of types σ_j
- x_i will be bound to values, as usual, of types τ_i

$$\frac{\Delta \;;\; \cdot \vdash e : \sigma}{\Delta \;;\; \Gamma \vdash \mathbf{box}\; e : \Box\sigma}\; \Box I \qquad \frac{\Delta \;;\; \Gamma \vdash e : \Box\sigma \quad \Delta, u{::}\sigma \;;\; \Gamma \vdash e' : \tau'}{\Delta \;;\; \Gamma \vdash \mathbf{let\ box}\; u = e\; \mathbf{in}\; e' : \tau'}\; \Box E$$

$$\frac{x : \tau \in \Gamma}{\Delta \;;\; \Gamma \vdash x : \tau}\; \text{hyp} \qquad \frac{u :: \sigma \in \Delta}{\Delta \;;\; \Gamma \vdash u : \sigma}\; \text{vhyp}$$

Dynamics

- Stepping rules (selection) correspond to proof reductions

  $\mathbf{box}\ e\ \mathit{value}$

  $\mathbf{let\ box}\ u = \mathbf{box}\ e\ \mathbf{in}\ e' \mapsto [\![e/u]\!]e'$

- Observations on substitution

  $[v/x]\,\mathbf{box}\ e' = \mathbf{box}\ e'$

  $[\![e/u]\!]\ \mathbf{box}\ e' = \mathbf{box}\ ([\![e/u]\!]\ e')$

- Satisfies the usual substitution and type soundness properties
- Typing guarantees proper staging

Example Revisited

- Quote the code!

  $power : \mathsf{nat} \to \Box(\mathsf{nat} \to \mathsf{nat})$
  $power = \lambda e.\ \ \textbf{if } e = 0 \textbf{ then box } (\lambda b.\, 1)$
  $\qquad\qquad\qquad\qquad \textbf{else let box } f = power\ (e - 1) \textbf{ in box } (\lambda b.\, b * f\ b)$

  $power\ 0 \mapsto^* \textbf{box } (\lambda b_0.\, 1)$
  $power\ 1 \mapsto^* \textbf{box } (\lambda b_1.\, b_1 * (\lambda b_0.\, 1)\ b_1)$
  $power\ 2 \mapsto^* \textbf{box } (\lambda b_2.\, b_2 * (\lambda b_1.\, b_1 * (\lambda b_0.\, 1)\ b_1)\ b_2)$
  $\qquad\qquad =_{\beta_v} \textbf{box } (\lambda b_2.\, b_2 * b_2 * 1)$

- Now $\Box\sigma$ is a compiled to a generator for code of type σ
- (Curried) functions are back to normal

Outline

- S4 and Runtime Code Generation
- Decomposing Modalities
- Lax Logic and Monadic Programming
- Adjoint Logic
- K and Partial Evaluation
- Substructural Adjoint Logic

Oddities in a Sequent Calculus

- Erasing terms yields a system of natural deduction
- In sequent calculus, we replace $u :: \sigma$ with $B\ \mathit{valid}$ and $x : \tau$ with $A\ \mathit{true}$

$$\underbrace{B_1\ \mathit{valid}, \ldots, B_m\ \mathit{valid}}_{\Delta} \;;\; \underbrace{A_1\ \mathit{true}, \ldots, A_n\ \mathit{true}}_{\Gamma} \vdash C\ \mathit{true}$$

- Show here some sequent calculus rules

$$\frac{\Delta \;;\; \cdot \vdash A\ \mathit{true}}{\Delta \;;\; \Gamma \vdash \Box A\ \mathit{true}}\ \Box R \qquad \frac{\Delta, A\ \mathit{valid} \;;\; \Gamma \vdash C\ \mathit{true}}{\Delta \;;\; \Box A\ \mathit{true}, \Gamma \vdash C\ \mathit{true}}\ \Box L$$

$$\frac{\Delta \;;\; \Gamma, A\ \mathit{true} \vdash C\ \mathit{true}}{\Delta, A\ \mathit{valid} \;;\; \Gamma \vdash C\ \mathit{true}}\ \mathrm{valid}L$$

- Some oddities:
  - The proposition A does not change in valid$L$
  - There is no valid$R$

Decomposing Box

- We stratify the syntax [Benton 1994] [Reed 2009]

$$\begin{array}{llll} \text{Validity} & V & ::= & \uparrow A \\ \text{Truth} & A & ::= & A_1 \to A_2 \mid A_1 \times A_2 \mid \ldots \mid \downarrow V \end{array}$$

- Judgments $A\ \mathit{true}$ and $V\ \mathit{valid}$
- Independence principle: validity cannot depend on truth

$$\frac{\Delta \vdash V\ \mathit{valid}}{\Delta\ ;\ \Gamma \vdash \downarrow V\ \mathit{true}}\ \downarrow R \qquad \frac{\Delta, V\ \mathit{valid}\ ;\ \Gamma \vdash C\ \mathit{true}}{\Delta\ ;\ \Gamma, \downarrow V\ \mathit{true} \vdash C\ \mathit{true}}\ \downarrow L$$

$$\frac{\Delta\ ;\ \cdot \vdash A\ \mathit{true}}{\Delta \vdash \uparrow A\ \mathit{valid}}\ \uparrow R \qquad \frac{\Delta\ ;\ \Gamma, A\ \mathit{true} \vdash C\ \mathit{true}}{\Delta, \uparrow A\ \mathit{valid}\ ;\ \Gamma \vdash C\ \mathit{true}}\ \uparrow L$$

$$\frac{}{\Delta\ ;\ \Gamma, A\ \mathit{true} \vdash A\ \mathit{true}}\ \mathrm{id} \qquad \frac{}{\Delta, V\ \mathit{valid} \vdash V\ \mathit{valid}}\ \mathrm{id}_v$$

Symmetry and Beauty Restored

- We can define $\Box A \triangleq \downarrow\uparrow A$
- $\uparrow$ and $\downarrow$ form are adjoint
- $\Box A$ is a comonad
- We can populate the validity layer!

$$\begin{array}{llll} \text{Validity} & V & ::= & V_1 \to V_2 \mid V_1 \times V_2 \mid \ldots \mid \uparrow A \\ \text{Truth} & A & ::= & A_1 \to A_2 \mid A_1 \times A_2 \mid \ldots \mid \downarrow V \end{array}$$

- The left and right rules for the validity layer are identical to those in the truth layer (except for $\uparrow$ and $\downarrow$)
  - We reason the same way in both layers
  - Modal logic: the meaning of the logical connectives does not depend on the world we are in

Outline

- S4 and Runtime Code Generation
- Decomposing Modalities
- Lax Logic and Monadic Programming
- Adjoint Logic
- K and Partial Evaluation
- Substructural Adjoint Logic

Lax Logic

- Now we can depopulate the truth layer

$$\begin{array}{lrcl} \text{Validity} & V & ::= & V_1 \to V_2 \mid V_1 \times V_2 \mid \ldots \mid \uparrow A \\ \text{Truth} & A & ::= & \downarrow V \end{array}$$

- Define $\bigcirc V \triangleq \uparrow\downarrow V$
- Since $\uparrow$ and $\downarrow$ are adjoint, this forms a (strong) monad
- This is now lax logic!
  [Fairtlough & Mendler 1994] [Benton, Bierman, de Paiva 1998]
- Relabel the layers

$$\begin{array}{lrcl} \text{Truth} & A & ::= & A_1 \to A_2 \mid A_1 \times A_2 \mid \ldots \mid \uparrow L \\ \text{Lax Truth} & L & ::= & \downarrow A \end{array}$$

- L *lax* means that L is true under some constraints
- Independence principle: Truth may not depend on lax truth!

Monadic Programming

- Briefly return to natural deduction
- We write $c \div \tau$ for c has lax type τ
- Still have $e : \sigma$ if e has type σ
- No assumption $w \div \tau$ can ever be introduced, so we only have value variables $x : \sigma$
- Slight variant of the usual bind/return

$$\frac{\Delta \vdash c \div \tau}{\Delta \vdash \mathbf{box}\ c : \uparrow\tau}\ \uparrow I \qquad \frac{\Delta \vdash e : \uparrow\tau}{\Delta \vdash \mathbf{unbox}\ e \div \tau}\ \uparrow E$$

$$\frac{\Delta \vdash e : \sigma}{\Delta \vdash \mathbf{return}\ e \div \downarrow\sigma}\ \downarrow I \qquad \frac{\Delta \vdash c \div \downarrow\sigma \quad \Delta, x : \sigma \vdash c' \div \tau}{\Delta \vdash \mathbf{let\ return}\ x = c\ \mathbf{in}\ c' \div \tau}\ \downarrow E_1$$

$$\frac{\Delta \vdash c \div \downarrow\sigma \quad \Delta, x : \sigma \vdash e' : \sigma}{\Delta \vdash \mathbf{let\ return}\ x = c\ \mathbf{in}\ e' : \sigma}\ \downarrow E_2$$

Ghost Code

- We can fully populate the lax layer to represent "ghost code"
- Used for proof/verification
- In type theory, this modeling proof irrelevance

$$\begin{array}{lllll} \text{Observable} & \sigma & ::= & \sigma_1 \to \sigma_2 \mid \sigma_1 \times \sigma_2 \mid \ldots \mid \uparrow\tau \\ \text{Ghost} & \tau & ::= & \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \ldots \mid \downarrow\sigma \end{array}$$

- Programs in ghost layer do not affect observables, by independence principle
- Ghost code can be erased

Combining Quotations and Ghosts

- We need at least two set of shifts!

$$
\begin{array}{lcll}
\text{Quoted} & \rho & ::= & \rho_1 \to \rho_2 \mid \rho_1 \times \rho_2 \mid \ldots \mid \Uparrow \sigma \\
\text{Extensional} & \sigma & ::= & \sigma_1 \to \sigma_2 \mid \sigma_1 \times \sigma_2 \mid \ldots \mid \uparrow \tau \mid \Downarrow \rho \\
\text{Ghost} & \tau & ::= & \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \ldots \mid \downarrow \sigma
\end{array}
$$

- Special case in type theory [Pf 2001]
- Generalize further to a set of modes?

Outline

- S4 and Runtime Code Generation
- Decomposing Modalities
- Lax Logic and Monadic Programming
- Adjoint Logic
- K and Partial Evaluation
- Substructural Adjoint Logic

Adjoint Logic

- [Reed 2009] [Chargin, Pf, Pruiksma, Reed 2020]
- Each proposition has an intrinsic mode m of truth
- We have a preorder $m \geq k$
  - Proof of A_k may depend on hypothesis A_m only if $m \geq k$
  - Contexts $\Psi ::= \cdot \mid \Psi, A_m$ (modes can be distinct)
  - $\Psi \geq k$ if $m \geq k$ for every $A_m \in \Psi$
- $\Psi \vdash A_k$ presupposes $\Psi \geq k$
- For $\ell \geq m \geq k$

$$\text{Props} \quad A_m \quad ::= \quad A_m \to B_m \mid A_m \times B_m \mid \ldots \mid \uparrow_k^m A_k \mid \downarrow_m^\ell A_\ell$$

- If $m > k$ then k may depend on m, but m is independent of k
- For example, valid $>$ true $>$ lax.

Adjoint Logic, Continued

- Rules for the connectives are uniform in the mode
- Language at a mode may have only subset of the connectives
- Present as a sequent calculus
  - Natural deduction also works
  - Axiomatic system also exists
- Adjoint logic as "module calculus" for combining logics
- Satisfies cut elimination once and for all

Rules of Adjoint Logic

$$\frac{}{A_m \vdash A_m}\ \mathrm{id} \qquad \frac{\Psi_1 \vdash A_m \quad \Psi_2, A_m \vdash C_k \quad (\Psi_1 \geq m \geq k)}{\Psi_1, \Psi_2 \vdash C_k}\ \mathrm{cut}$$

$$\frac{\Psi, A_m \vdash B_m}{\Psi \vdash A_m \to B_m} \to R \qquad \frac{\Psi_1 \vdash A_m \quad \Psi_2, B_m \vdash C_k \quad (\Psi_1 \geq m)}{\Psi_1, \Psi_2, A_m \to B_m \vdash C_k} \to L$$

$$\frac{\Psi \vdash A_k}{\Psi \vdash \uparrow_k^m A_k} \uparrow R \qquad \frac{\Psi, A_k \vdash C_\ell \quad (k \geq \ell)}{\Psi, \uparrow_k^m A_k \vdash C_\ell} \uparrow L$$

$$\frac{\Psi \vdash A_\ell \quad (\Psi \geq A_\ell)}{\Psi \vdash \downarrow_m^\ell A_\ell} \downarrow R \qquad \frac{\Psi, A_\ell \vdash C_k}{\Psi, \downarrow_m^\ell A_\ell \vdash C_k} \downarrow L$$

$$\frac{\Psi \vdash C_k}{\Psi, A_m \vdash C_k}\ \mathrm{weaken} \qquad \frac{\Psi, A_m, A_m \vdash C_k}{\Psi, A_m \vdash C_k}\ \mathrm{contract}$$

Outline

- S4 and Runtime Code Generation
- Decomposing Modalities
- Lax Logic and Monadic Programming
- Adjoint Logic
- K and Partial Evaluation
- Substructural Adjoint Logic

Back to Partial Evaluation

- Rowan Davies. *A Temporal-Logic Approach to Binding Time Analysis*, LICS 1996.
- Modes are binding times $0, 1, 2, \ldots$
- Each layer populated with all logical connectives
- We only have the down shifts $\downarrow_t^{t+1} A_{t+1}$ (generically $\downarrow A$)
  - Represents the next-time operator $\bigcirc A \triangleq \downarrow_t^{t+1} A_{t+1}$
- First, sequent calculus

$$\frac{\Psi \vdash A_{t+1}}{\Psi \vdash \downarrow A_{t+1}} \downarrow R \qquad \frac{\Psi, A_{t+1} \vdash C_s}{\Psi, \downarrow A_{t+1} \vdash C_s} \downarrow L$$

- Mode structure $t \geq s$ for all t and s (others possible)
- Essentially, we obtain the modal logic K with only

$$\downarrow(A \to B) \to (\downarrow A \to \downarrow B) \qquad (\text{dist} \to)$$

- For example, we cannot define eval

Terms and Operational Semantics

- Switch to natural deduction

$$\frac{\Psi \vdash e : \tau_{t+1}}{\Psi \vdash \textbf{next}\ e : \downarrow\tau_{t+1}}\ \downarrow I \qquad \frac{\Psi \vdash e : \downarrow\tau_{t+1}}{\Psi \vdash \textbf{prev}\ e : \tau_{t+1}}\ \downarrow E$$

- Step only at time 0, postpone others to future

$$\frac{}{(\lambda x.\, e)\ v \mapsto_0 [v/x]e} \qquad \frac{}{(\lambda x.\, e)\ value_0}$$

$$\frac{e \mapsto_{t+1} e'}{\textbf{next}\ e \mapsto_t \textbf{next}\ e'} \qquad \frac{e \mapsto_t e'}{\textbf{prev}\ e \mapsto_{t+1} \textbf{prev}\ e'}$$

$$\frac{e \mapsto e'}{\lambda x.\, e \mapsto_{t+1} \lambda x.\, e'} \qquad \frac{}{x \mapsto_{t+1} x} \qquad \frac{e_1 \mapsto_{t+1} e_1' \quad e_2 \mapsto_{t+1} e_2'}{e_1\ e_2 \mapsto_{t+1} e_1'\ e_2'}$$

Example Revisited

- Canonical forms: if $\cdot \vdash v : \downarrow \tau_1$ and v *value*$_0$ then $v = \textbf{next}\ e$ and $\cdot \vdash e : \tau_1$
- We proceed in stages, globally
- Evaluate underneath abstractions (but only reduce at time 0)

$$
\begin{array}{l}
\mathit{power} : \mathsf{nat} \to \downarrow(\mathsf{nat} \to \mathsf{nat}) \\
\mathit{power} = \lambda e.\, \textbf{next}\ \lambda b.\, \textbf{prev} \\
\qquad\qquad (\mu p.\, \lambda n.\ \ \textbf{if}\ n = 0\ \textbf{then next}\ 1 \\
\qquad\qquad\qquad\qquad\ \ \textbf{else next}\ (b * \textbf{prev}\ (p\ (n - 1)))) \\
\qquad\qquad e
\end{array}
$$

$$
\begin{array}{l}
\mathit{power}\ 0 \mapsto^* \textbf{next}\ (\lambda b.\, 1) \\
\mathit{power}\ 1 \mapsto^* \textbf{next}\ (\lambda b.\, b * 1) \\
\mathit{power}\ 2 \mapsto^* \textbf{next}\ (\lambda b.\, b * b * 1)
\end{array}
$$

Outline

- S4 and Runtime Code Generation
- Decomposing Modalities
- Lax Logic and Monadic Programming
- Adjoint Logic
- K and Partial Evaluation
- Substructural Adjoint Logic

Linear Logic

- There are three related origins of adjoint logic
  - Nick Benton. *A Mixed Linear and Non-Linear Logic*, CSL 1994.
  - Vivek Nigam & Dale Miller. *Algorithmic Specifications in Linear Logic with Subexponentials*, PPDP 2009.
  - Jason Reed. *A Judgmental Deconstruction of Modal Logic*, 2009.
- Benton: Two modes (linear and nonlinear) with shifts
- Nigam & Miller: a preorder of substructural modes with exponentials
- Reed: modal logic with a preorder of modes and shifts
- Intuitionistic linear logic, with U $>$ L

$$\begin{array}{llll} \text{Nonlinear} & A_{\mathsf{U}} & ::= & \uparrow A_{\mathsf{L}} \\ \text{Linear} & A_{\mathsf{L}} & ::= & A_{\mathsf{L}} \multimap B_{\mathsf{L}} \mid A_{\mathsf{L}} \otimes B_{\mathsf{L}} \mid \ldots \mid \downarrow A_{\mathsf{U}} \end{array}$$

- Define $!A_{\mathsf{L}} \triangleq \downarrow\uparrow A_{\mathsf{L}}$
- Benton's LNL populates the nonlinear layer

$$\begin{array}{llll} \text{Nonlinear} & A_{\mathsf{U}} & ::= & A_{\mathsf{U}} \to B_{\mathsf{U}} \mid A_{\mathsf{U}} \times B_{\mathsf{U}} \mid \ldots \mid \uparrow A_{\mathsf{L}} \\ \text{Linear} & A_{\mathsf{L}} & ::= & A_{\mathsf{L}} \multimap B_{\mathsf{L}} \mid A_{\mathsf{L}} \otimes B_{\mathsf{L}} \mid \ldots \mid \downarrow A_{\mathsf{U}} \end{array}$$

Generalize to Substructural Adjoint Logic

- Each mode has a set of structural properties $\sigma(m) \subseteq \{W, C\}$ among Weakening and Contraction
- Requires: If $m \geq k$ then $\sigma(m) \supseteq \sigma(k)$
- Obtain general cut elimination
- Only two rules change

$$\frac{\Psi \vdash C_k \quad (W \in \sigma(m))}{\Psi, A_m \vdash C_k}\ \text{weaken} \qquad \frac{\Psi, A_m, A_m \vdash C_k \quad (C \in \sigma(m))}{\Psi, A_m \vdash C_k}\ \text{contract}$$

- Substructural adjoint logic generalizes subexponentials [Nigam & Miller 2009] [Chaudhuri 2010]
  - Allow modes as in adjoint logic; distinguished mode L
  - The modalities $!^m A_{\mathsf{L}}$ are substructural
  - Decomposed to $!^m A_{\mathsf{L}} \triangleq \downarrow^m_{\mathsf{L}} \uparrow^m_{\mathsf{L}} A_{\mathsf{L}}$

Substructural Adjoint Logic

- Integration of layers with different structural properties
- General cut elimination
- Conservative extension over each mode
- Computational interpretations
  - Example: concurrent programming with linear and nonlinear futures [Blelloch & Reid-Miller 1999] [Pruiksma & Pf 2020]
  - Example: message passing concurrency with multicast and persistent servers [Pruiksma & Pf 2021]
- Modal/linear logics: Encode validity/nonlinear reasoning via $\Box A$ or $!A$
- Substructural adjoint logic: Reason natively in each logic and switch between them

Summary

- Then [1994–2000]:
  - $\Box A$: Modal logic S4
  - Intensionality and runtime code generation
  - $\bigcirc A$: Lax logic
  - Ghosts and monadic programming with effects
  - **Next** A: Modal logic K
  - Partial evaluation
  - $F\,X$ and $G\,A$: Mixed linear/nonlinear logic
- In between [2000-2018]:
  - Modal types as a standard tool in programming languages
  - Subexponentials
- Now [2018–2022]
  - $\downarrow_m^\ell A_\ell$ and $\uparrow_k^m A_k$: Substructural adjoint logic
  - Key ingredient: independence principle
  - All of the above are instances of the same framework
  - All of the above can be coherently combined

Some Puzzles

- Possibility ($\Diamond A$) vs. lax truth ($\bigcirc$ A)
  [Reed 2009] [Licata & Shulman 2016]
- Polarization [Levy 2001] [Laurent 2003]
  - Looks the same, but how do we best integrate or relate it
  - For example, $A^+ \to B^-$ combines distinct polarities/modes
- Mode-generic programming
- From simple to contextual and dependent types
- From modular language to modular reasoning [Pruiksma 2022]
- More structure in shift modalities [Licata & Shulman 2016]

Thanks!

Seminal work
Nick Benton, Vivek Nigam & Dale Miller, Jason Reed

Collaborators on this and closely related work
Rowan Davies,
Iliano Cervesato, David Walker, Kevin Watkins, Aleks Nanevski, Brigitte Pientka, Dennis Griffith, William Chargin, Stephanie Balzer, Aditi Gupta,
Klaas Pruiksma