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

Terminology

- **Staged Computation:** explicit division of a computation into stages. Used in algorithm derivation and program optimization.

- **Partial Evaluation:** (static) specialization of a program based on partial input data.

- **Run-Time Code Generation:** dynamic generation of code during the evaluation of a program.

Intensionality

- Staged computation is concerned with **how** a value is computed.

- Staging is an **intensional** property of a program.

- Most research has been motivated **operationally**.

- This talk: a **logical** way to understand staging which is consistent with the operational intuition.
  [Davies & Pf. POPL'96] [Davies & Pf.'99]

Logical Foundations for Computation

- Specifications as Propositions as Types
- Implementations as Proofs as Programs
- Computations as Reductions as Evaluations
- Augmented by recursion, exceptions, effects, ...

Judgments and Propositions [Martin-Löf]

- A *judgment* is an object of knowledge.

- An *evident judgment* is something we know.

- The meaning of a *proposition* A is given by what counts as a verification of A.

- A is *true* if there is a proof M of A.

- Basic judgment: $M : A$.

Parametric and Hypothetical Judgments

- Parametric and hypothetical judgments

$$\underbrace{x_1{:}A_1, \ldots, x_n{:}A_n}_{\Gamma} \vdash M : A$$

- Meaning given by **substitution**

  If $\Gamma, x{:}A \vdash N : C$
  and $\Gamma \vdash M : A$
  them $\Gamma \vdash [M/x]N : C$

- Order in Γ irrelevant, satisfies weakening and contraction.

- Hypothesis or variable rule

$$\frac{}{\Gamma, x{:}A \vdash x : A}\,\mathsf{var}$$

Implication and Function Types

- Reflecting a hypothetical judgment as a proposition.

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\, M : A \rightarrow B} \rightarrow I$$

$$\frac{\Gamma \vdash M : A \rightarrow B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B} \rightarrow E$$

- How do we know these rules are consistent?
- Martin-Löf's *meaning explanation*.
- Summarize as local soundness and completeness.

Local Soundness

- *Local soundness*: the elimination rules are not too strong.
- An introduction rule followed by any elimination rule does not lead to new knowledge.
- Witnessed by *local reduction*

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \Gamma, x{:}A \vdash M : B\end{array}}{\Gamma \vdash (\lambda x{:}A.\, M) : A \rightarrow B}{\rightarrow}I \qquad \begin{array}{c}\mathcal{E}\\ \Gamma \vdash N : A\end{array}}{\Gamma \vdash (\lambda x{:}A.\, M)\, N : B}{\rightarrow}E \quad \Longrightarrow_R \quad \begin{array}{c}\mathcal{D}'\\ \Gamma \vdash [N/x]M : B\end{array}$$

- $\mathcal{D}'$ exists by the substitution property of hypothetical judgments.

Local Completeness

- *Local completeness*: the elimination rules are not too weak.

- We can apply the elimination rules in such a way that a derivation of the original judgment can be reconstituted from the results.

- Witnessed by *local expansion*

$$
\begin{array}{c}\mathcal{D}\\ \Gamma \vdash M : A \to B\end{array}
\quad \Longrightarrow_E \quad
\dfrac{\dfrac{\begin{array}{c}\mathcal{D}'\\ \Gamma, x{:}A \vdash M : A \to B\end{array} \qquad \dfrac{}{\Gamma, x{:}A \vdash x{:}A}\,\text{var}}{\Gamma, x{:}A \vdash M\,x : B}\,{\to}E}{\Gamma \vdash (\lambda x{:}A.\, M\,x) : A \to B}\,{\to}I
$$

- $\mathcal{D}'$ exists by weakening.

Reduction and Evaluation

- Reduction: $(\lambda x{:}A.\, M)\, N \Longrightarrow_R [N/x]M$ at any subterm.
- Local soundness means reduction preserves types.
- Evaluation = reduction + strategy (here: call-by-value)

$$\text{Values } V ::= \lambda x{:}A.\, M \mid \ldots$$

$$\frac{}{\lambda x{:}A.\, M \hookrightarrow \lambda x{:}A.\, M}$$

$$\frac{M \hookrightarrow \lambda x{:}A.\, M' \qquad N \hookrightarrow V' \qquad [V'/x]M' \hookrightarrow V}{M\, N \hookrightarrow V}$$

Towards Functional Programming

- Decide on *observable types*.

- Functions are not observable
  — allows us to compile and optimize.

- Functions are extensional
  — we can determine their behavior on arguments, but not their definition.

- Evaluate M only if $\cdot \vdash M : A$.

- If $x_1{:}A_1, \ldots, x_n{:}A_n \vdash M : A$ then we may evaluate $[V_1/x_1, \ldots, V_n/x_n]M$.

Logical Foundations for *Staged* Computation

- *Staging* Specifications (as Propositions as Types)

- *Staged* Implementations (as Proofs as Programs)

- *Staged* Computations (as Reductions as Evaluations)

- Augmented by recursion, exceptions, effects, ...

Desirable Properties

- Local soundness and completeness.
- Evaluation preserves types.
- Conservative extension (orthogonality).
- Captures staging.

Some Design Principles

- Explicit: put the power of staging in the hands of the programmer, not the compiler.

- Static: staging errors should be type errors.

- Implementable: can achieve expected efficiency improvements.

Focus: Run-Time Code Generation

- Generate code for portions of the program at run-time to take advantage of information only available then.

- Examples: sparse matrix multiplication, regular expression matchers, . . .

- Implementation via code generators or templates.

Requirements

- To “compile” at run-time we need a source expression.
- Enable optimizations, but do not force them.
- Distinguish *terms* from *source expressions*.
- The structure of (functional) terms is **not** observable: **extensional**.
- The structure of source expressions may be observable: **intensional**.

Categorical Judgments

- $M :: A$ — M is a *source expression* of type A.

- Do not duplicate constructors or types.

- Instead define: M is a source expression if it does not depend on any (extensional) terms.

$$\vdash M :: A \quad \textit{if} \quad \cdot \vdash M : A$$

- A is *valid* (categorically true)
  if A has a proof which does not depend on hypotheses.

Generalized Hypothetical Judgments

- Generalize to permit hypotheses $u{::}B$.

$$\underbrace{u_1{::}B_1, \ldots, u_m{::}B_m}_{\Delta}; \underbrace{x_1{:}A_1, \ldots x_n{:}A_n}_{\Gamma} \vdash M : A$$

- Meaning given by substitution

If $(\Delta, u{::}B); \Gamma \vdash N : C$
and $\Delta; \cdot \vdash M : B$ *(i.e.,* $\Delta \vdash M :: B$*)*
then $\Delta; \Gamma \vdash [\![M/u]\!]N : C$

- New hypothesis rule

$$\frac{}{(\Delta, u{::}B); \Gamma \vdash u : B}\ \text{var*}$$

Reflection

- $\Box A$ — proposition expressing that A is valid.

- $M : \Box A$ — M is a term which stands for (evaluates to) a source expression of type A.

- Introduction rule.

$$\frac{\Delta; \cdot \vdash M : A}{\Delta; \Gamma \vdash \mathbf{box}\, M : \Box A} \Box I$$

- Premise expresses
  A is valid, or
  M is a source expression of type A.

Elimination Rule

- Attempt:

$$\frac{\Delta; \Gamma \vdash M : \Box A}{\Delta; \Gamma \vdash \mathbf{unbox}\, M : A}\ \Box E??$$

- Locally sound (by weakening):

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \Delta; \cdot \vdash M : A\end{array}}{\Delta; \Gamma \vdash \mathbf{box}\, M : \Box A}\ \Box I}{\Delta; \Gamma \vdash \mathbf{unbox}\,(\mathbf{box}\, M) : A}\ \Box E \quad \Longrightarrow_R \quad \begin{array}{c}\mathcal{D}'\\ \Delta; \Gamma \vdash M : A\end{array}$$

- Definable later: eval : $(\Box A) \to A$.

Failure of Local Completeness

- Elimination rule is too weak.

- **Not** locally complete: $M{:}\Box A \Longrightarrow_E$?? **box** (**unbox** M).

$$\begin{array}{c}\mathcal{D}\\ \Delta;\Gamma\vdash M:\Box A\end{array} \Longrightarrow_E \dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \Delta;\Gamma\vdash M:\Box A\end{array}}{\Delta;\Gamma\vdash \mathbf{unbox}\,M:A}\,\Box E}{\Delta;\Gamma\vdash \mathbf{box}\,(\mathbf{unbox}\,M):\Box A}\,\Box I??$$

- Also cannot prove: $\vdash \Box(A \to B) \to \Box A \to \Box B$.

Elimination Rule Revisited

- Elimination rule

$$\frac{\Delta; \Gamma \vdash M : \Box A \qquad (\Delta, u{::}A); \Gamma \vdash N : C}{\Delta; \Gamma \vdash \textbf{let box}\, u = M\, \textbf{in}\, N : C}\,\Box E$$

- Locally sound

$$\frac{\dfrac{\begin{array}{c}\mathcal{D}\\ \Delta; \cdot \vdash M : A\end{array}}{\Delta; \Gamma \vdash \textbf{box}\, M : \Box A}\,\Box I \qquad \begin{array}{c}\mathcal{E}\\ (\Delta, u{::}A); \Gamma \vdash N : C\end{array}}{\Delta; \Gamma \vdash \textbf{let box}\, u = \textbf{box}\, M\, \textbf{in}\, N : C}\,\Box E$$

$$\Longrightarrow_R \quad \begin{array}{c}\mathcal{E}'\\ \Delta; \Gamma \vdash [\![M/u]\!]N : C\end{array}$$

Local Completeness

- Local expansion

$$\begin{array}{c}\mathcal{D}\\ \Delta;\Gamma \vdash M : \Box A\end{array}$$

$$\Longrightarrow_E \quad \dfrac{\begin{array}{c}\mathcal{D}\\ \Delta;\Gamma \vdash M : \Box A\end{array} \qquad \dfrac{\dfrac{}{(\Delta, u{::}A);\cdot \vdash u : A}\,\text{var*}}{(\Delta, u{::}A);\Gamma \vdash \mathbf{box}\, u : \Box A}\,\Box I}{\Delta;\Gamma \vdash (\mathbf{let\ box}\, u = M\, \mathbf{in}\, \mathbf{box}\, u) : \Box A}\,\Box E$$

- On terms:

$$M : \Box A \Longrightarrow_E \mathbf{let\ box}\, u = M\, \mathbf{in}\, \mathbf{box}\, u$$

Summary of Reductions

- Reductions as basis for operational semantics.
- $(\lambda x{:}A.\, M)\, N \Longrightarrow_R [N/x]M$
- $\textbf{let box}\, u = \textbf{box}\, M\, \textbf{in}\, N \Longrightarrow_R [\![M/u]\!]N$
- Expansions as extensionality principles.
- $M : A \rightarrow B \Longrightarrow_E (\lambda x{:}A.\, M\, x)$
- $M : \Box A \Longrightarrow_E (\textbf{let box}\, u = M\, \textbf{in}\, \textbf{box}\, u)$.

Some Examples

- Application

$$\cdot \;\vdash\; \lambda x{:}\Box(A \to B).\, \lambda y{:}\Box A.\; \textbf{let box}\, u = x\, \textbf{in}\, \textbf{let box}\, w = y\, \textbf{in}\, \textbf{box}\,(u\, w)$$

$$: \Box(A \to B) \to \Box A \to \Box B$$

- Evaluation

$$\cdot \;\vdash\; \lambda x{:}\Box A.\, \textbf{let box}\, u = x\, \textbf{in}\, u$$

$$: \Box A \to A$$

- Quotation

$$\cdot \;\vdash\; \lambda x{:}\Box A.\, \textbf{let box}\, u = x\, \textbf{in}\, \textbf{box}\,(\textbf{box}\, u)$$

$$: \Box A \to \Box\Box A$$

Logical Assessment

- $\Box$ satisfies laws of intuitionistic S_4.
- Cleaner and simpler formulation through judgmental reconstruction.
- Can be extended to capture $\Diamond$.
- (An aside: model Moggi's computational meta-language

$\Box A$	Value of type A
$\Diamond\Box A$	Computation of type A
$\Diamond\Box A$	$= \bigcirc A$ of lax logic

)

Operational Semantics

- Values $\lambda x{:}A.\, M$ and $\mathbf{box}\, M$.

- Rules

$$\frac{}{\mathbf{box} M \hookrightarrow \mathbf{box} M}$$

$$\frac{M \hookrightarrow \mathbf{box}\, M' \qquad [\![M'/u]\!]N \hookrightarrow V}{(\mathbf{let\ box}\, u = M\, \mathbf{in}\, N) \hookrightarrow V}$$

- $\mathbf{box}\, M$ may or may not be observable since M is guaranteed to be a source expression even if functions are compiled.

- Fully compatible with recursion, effects.

Desirable Properties Revisited

- Local soundness and completeness. **yes**
- Evaluation preserves types. **yes**
- Conservative extension (orthogonality). **yes**
- Captures staging.
  captures intensional expressions reflectively
- Enables, but does not force optimizations.

Observable Intensional Types

- Source expressions must be manipulated explicitly during computation.

- Source expressions are evaluated in contexts
$$\textbf{let box}\, u = M\, \textbf{in}\; \ldots u \ldots$$
where u is not inside a **box** constructor.

- Source expression could be interpreted, or compiled and then executed.

- A **case** construct for source expressions(!) which does not violate α-conversion can be added safely.
[Despeyroux, Schürmann, Pf. TLCA'97] [Schürmann & Pf. CADE'98] [Pitts & Gabbay '00]

Some Applications

- Type-safe macros
- Meta-programming
- Symbolic computation
- (An aside: Mathematica does not distinguish $\textbf{box}\,(2^{2^{2^{2^{2}}}} - 1)$ and $2^{2^{2^{2^{2}}}} - 1$, but should!)

Non-Observable Intensional Types

- Obtain a pure system of run-time code generation.
- We may compile **box** M to a *code generator*.
- This generator is a function of its free expression variables u_j (value variables x_i cannot occur free in M!)
- Implemented in the PML compiler (in progress).

The PML Language

- [Wickline, Lee, Pfenning PLDI'98] (in progress)
- Core ML (recursion, data types, mutable references) extended by types $\Box A$ (written `[A]`).
- Lift for observable types (similar to equality types).
- Staging errors are type errors (but ...).
- Memoization must be programmed explicitly.

Structure of the Compiler

- Standard parsing, type-checking.
- “Split” (2-environment) closure conversion.
- Standard ML-RISC code generator for unstaged code.
- Lightweight run-time code generation (Fabius [Lee & Leone’96]).

Closed Code Generators

- Compiling **box** M where M is closed.
- Compile M obtaining binary B (using ML-RISC).
- Write code C to generate B.
- Generate binary for **box** M from C (using ML-RISC).
- Backpatching for forward jumps and branches at code generation time (run-time system).

Open Code Generators

- Compiling **let box** $u = N$ **in** ... **box** M ...
- At run-time, u will be bound to a code generator.
- The generator for M will call the generator u.
- Planned: pass register information (right now: standard calling convention).
- Planned: type-based optimization at interface (Fabius).

Nested Code Generators

- Special treatment for nested code generators to avoid code explosion.

- Conceptually:

$$\begin{array}{rcl} \textbf{box}\, M & \mapsto & \lambda x{:}\mathsf{unit}.\, M \\ \textbf{let box}\, u = M \,\textbf{in}\, N & \mapsto & \textbf{let val}\, x = M \,\textbf{in}\, [x\,()/u]N \end{array}$$

- Observationally equivalent, but prohibits any optimizations.

Invoking Generated Code

- Compiling **let box** $u = N$ **in** $\ldots u \ldots$, u not "boxed".
- Call code generator for u.
- Jump to generated code.

Example: Regular Expression Matcher

```
datatype regexp
   = Empty                          (* e       empty string  *)
   | Plus of regexp * regexp        (* r1 + r2 union         *)
   | Times of regexp * regexp       (* r1 r2   concatenation *)
   | Star of regexp                 (* r*      iteration     *)
   | Const of string                (* a       letter        *)
(* aux function *)
val acc : regexp -> (string list -> bool)
           -> (string list -> bool)
```

acc r k s $\hookrightarrow$ **true**
iff $s = s_1 @ s_2$ where $s_1 \in \mathcal{L}(r)$ and k $s_2 \hookrightarrow$ **true** for some s_1 and s_2.

```
fun accept r s = acc r List.null s
```

Unstaged Implementation

```
fun acc (Empty) k s = k s
  | acc (Plus(r1,r2)) k s =
      acc r1 k s orelse acc r2 k s
  | acc (Times(r1,r2)) k s =
      acc r1 (fn ss => acc r2 k ss) s
  | acc (Star(r)) k s =
      k s orelse
      acc r (fn ss => if s = ss then false
                      else acc (Star(r)) k ss) s
  | acc (Const(str)) k (x::s) =
      (x = str) andalso k s
  | acc (Const(str)) k (nil) = false
```

Staged Version, Part I

```
(* val acc : regexp ->
    [(string list -> bool) -> (string list -> bool)] *)
fun acc (Empty) = box (fn k => fn s => k s)
  ...
  | acc (Times(r1,r2)) =
    let box a1 = acc r1
        box a2 = acc r2
    in
        box (fn k => fn s => a1 (fn ss => a2 k ss) s)
    end
  | acc (Star(r1)) =
    let box a1 = acc r1
        box rec aStar =
         box (fn k => fn s =>
              k s orelse
              a1 (fn ss => if s = ss then false
                           else aStar k ss) s)
    in
        box (fn k => fn s => aStar k s)
    end
```

Staged Version, Part II

```
  | acc (Const(c)) =
    let box c' = lift c  (* c : string *)
    in
        box (fn k => (fn (x::s) => (x = c') andalso k s
                       | nil => false))
    end
(* val accept3 : regexp -> (string list -> bool) *)
fun accept3 r =
    let box a = acc r
    in
      a List.null
    end
```

Example

```
Times (Const "a", Empty)
=>
let box a1 =
     box (fn k => (fn (x::s) => (x = "a") andalso k s
                    | nil => false))
    box a2 = box (fn k => fn s => k s)
in
    box (fn k => fn s => a1 (fn ss => a2 k ss) s)
end
=>
box (fn k => fn s =>
      (fn k => (fn (x::s) => (x = "a") andalso k s
                   | nil => false))
      (fn ss => (fn k => fn s => k s) k ss) s)
```

A Sample Optimization

Substitute variable for variable, functional value for linear variable.

```
box (fn k => fn s =>
      (fn k => (fn (x::s) => (x = "a") andalso k s
                   | nil => false))
      (fn ss => (fn k => fn s => k s) k ss) s)
==>
box (fn k => fn s =>
      (fn (x::s') => (x = "a") andalso
                 (fn ss => (fn k => fn s => k s) k ss) s'
        | nil => false)) s)
==>
box (fn k => fn s =>
      (fn (x::s') => (x = "a") andalso k s'
        | nil => false)) s)
```

Run-Time Code Generation Summary

- Logical reconstruction yields clean and simple type system for run-time code generation.
- Application of Curry-Howard isomorphism to intuitionistic S_4.
- Distinguish expressions from terms (valid from true propositions).
- Enables optimizations without prescribing them.
- (Partially) implemented in the PML compiler.

Some Issues

- Lift for functions? Top-level? Modules?
- Memoization? Garbage collections for generated code?
- Some inference?
- Empirical study (cf. Fabius).

Implicit Syntax

- Derived (logically) from Kripke semantics of S_4.
- Similar to quasi-quote in Lisp-like languages.
- Operational semantics defined by translation.

```
fun acc (Empty) = `(fn k => fn s => k s)
  | acc (Times(r1,r2)) =
    `(fn k => fn s => ^(acc r1) (fn ss => ^(acc r2) k ss) s)
  | acc (Star(r1)) =
    `(fn k => fn s =>
      k s orelse
      ^(acc r1) (fn ss => if s = ss then false
                          else ^(acc (Star(r1))) k ss) s)
  ...
```

- Note bug!

Relation to Two-Level Languages

- Conservative extension of Nielson & Nielson [book version].
- Evident from implicit syntax.
- Allows arbitrary stages [Glück & Jørgensen PLILP'95].
- Two-level languages are one-level languages with modal types.

Relation to Partial Evaluation

- Partial evaluation *prescribes* optimization.
- Computation proceeds in discrete transformation steps.
- No analogue of eval : $\Box A \rightarrow A$.
- Logical foundations through intuitionistic linear time temporal logic. [Davies LICS'96]
- Combination subject to current research [Moggi, Taha, Benaissa, Sheard ESOP'99] [Davies & Pf.]
- Soundness problems in the presence of effects.

Conclusion

- Cleaner, simpler systems through judgmental analysis and logical foundation.
- Two-level languages are one-level languages with modal types.
- Put the power of the staged computation into the hands of the programmer, not the compiler!
- Staging errors should be type errors.