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

\[
\frac{x_1:A_1, \ldots, x_n:A_n}{\Gamma} \vdash M : A
\]

- Meaning given by substitution

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

- Order in \( \Gamma \) irrelevant, satisfies weakening and contraction.

- Hypothesis or variable rule

\[
\frac{}{\Gamma, x:A \vdash x : A} \text{ var}
\]
Implication and Function Types

- Reflecting a hypothetical judgment as a proposition.

\[
\begin{align*}
\Gamma, x: A &\vdash M : B \\
\rightarrow & \\
\Gamma &\vdash \lambda x: A. M : A \rightarrow B \\
\rightarrow & \text{I}
\end{align*}
\]

\[
\begin{align*}
\Gamma &\vdash M : A \rightarrow B \\
\Gamma &\vdash N : A \\
\rightarrow & \text{E}
\end{align*}
\]

- 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*

\[
\begin{align*}
&D \\
&\Gamma, x : A \vdash M : B \\
&\quad \Gamma \vdash (\lambda x : A. M) : A \rightarrow B \quad \rightarrow I \\
&\quad \Gamma \vdash N : A \\
&\quad \Gamma \vdash (\lambda x : A. M) N : B \quad \rightarrow E
\end{align*}
\]

- \( \mathcal{D}' \) 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*

\[
\begin{align*}
D & \quad \Rightarrow_E
\end{align*}
\]

\[
\Gamma \vdash M : A \rightarrow B
\]

\[
\begin{array}{c}
D'

d \Gamma, x:A \vdash M : A \rightarrow B
\end{array}
\]

\[
\begin{array}{c}
\Gamma, x:A \vdash x:A
\end{array}
\]

\[
\begin{array}{c}
\var
\end{array}
\]

\[
\begin{array}{c}
\rightarrow_E
\end{array}
\]

\[
\begin{array}{c}
\Gamma, x:A \vdash M x : B
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash (\lambda x:A. M \ x) : A \rightarrow B
\end{array}
\]

\[
\begin{array}{c}
\rightarrow_I
\end{array}
\]

- \(D'\) exists by weakening.
Reduction and Evaluation

- Reduction: \((\lambda x:A. M) N \overset{R}{\Rightarrow} [N/x]M\) at any subterm.

- Local soundness means reduction preserves types.

- Evaluation = reduction + strategy (here: call-by-value)

\[
\text{Values } V ::= \lambda x:A. M | \ldots
\]

\[
\begin{align*}
\lambda x:A. M & \leftrightarrow \lambda x:A. M \\
M & \leftrightarrow \lambda x:A. M' & N & \leftrightarrow V' & [V'/x]M' & \leftrightarrow V \\
M N & \leftrightarrow V
\end{align*}
\]
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 $\vdash M : A$.

- 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.

\[
\vdash M :: A \quad \text{if} \quad \vdash M : A
\]

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

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

\[
\frac{\Delta, u :: B_1, \ldots, u_m :: B_m; x_1:A_1, \ldots x_n:A_n}{\Gamma} \vdash M : A
\]

- Meaning given by substitution

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

- New hypothesis rule

\[
(\Delta, u :: B); \Gamma \vdash u : B \quad \text{var}^\ast
\]
Reflection

• \( \square A \) — proposition expressing that \( A \) is valid.

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

• Introduction rule.

\[
\Delta; \Gamma \vdash M : A \\
\hline
\Delta; \Gamma \vdash \text{box} M : \square A
\]

\( \text{\( \square I \)} \)

• Premise expresses

\( A \) is valid, or

\( M \) is a source expression of type \( A \).
Elimination Rule

- Attempt:

\[
\Delta; \Gamma \vdash M : \square A \\
\Delta; \Gamma \vdash \text{unbox } M : A \\
\square E?
\]

- Locally sound (by weakening):

\[
\begin{array}{c}
\Delta; \cdot \vdash M : A \\
\Delta; \Gamma \vdash \text{box } M : \square A \\
\Delta; \Gamma \vdash \text{unbox (box } M ) : A \\
\Delta; \Gamma \vdash M : A
\end{array}
\]

- Definable later: \( \text{eval} : (\square A) \rightarrow A \).
Failure of Local Completeness

- Elimination rule is too weak.

- **Not** locally complete: \( M : \square A \Rightarrow \Gamma \Rightarrow_E \square (\text{unbox } M) \).

\[
\frac{\Delta ; \Gamma \vdash M : \square A}{\frac{\Delta ; \Gamma \vdash \text{unbox } M : A}{\Delta ; \Gamma \vdash \text{box (unbox } M) : \square A}} \square I \?
\]

- Also cannot prove: \( \vdash \square (A \rightarrow B) \rightarrow \square A \rightarrow \square B \).
Elimination Rule Revisited

- Elimination rule

\[
\frac{\Delta; \Gamma \vdash M : \Box A}{\Delta; \Gamma \vdash \text{let } \text{box } u = M \text{ in } N : C} \quad \Box E
\]

- Locally sound

\[
\frac{\Delta; \vdash M : A \quad (\Delta, u :: A); \Gamma \vdash N : C}{\Delta; \Gamma \vdash \text{box } M : \Box A} \quad \Box I
\]

\[
\frac{\Delta; \Gamma \vdash \text{let } \text{box } u = \text{box } M \text{ in } N : C}{(\Delta, u :: A); \Gamma \vdash N : C} \quad \Box E
\]

\[
\implies_{R} \quad \frac{\Delta; \Gamma \vdash [M/u]N : C}{\Delta; \Gamma \vdash N : C} \quad \mathcal{E}'
\]
Local Completeness

- Local expansion

\[ \Delta; \Gamma \vdash M : \Box A \]

\[ \frac{\Delta; \Gamma \vdash M : \Box A \quad \frac{\Delta, u :: A; \cdot \vdash u : A}{(\Delta, u :: A); \Gamma \vdash \text{box } u : \Box A} \quad \text{var}^*}{\Delta; \Gamma \vdash (\text{let } \text{box } u = M \text{ in } \text{box } u) : \Box A} \quad I \]

\[ \frac{\Delta; \Gamma \vdash M : \Box A}{\Box E} \]

- On terms:

\[ M : \Box A \iff_E \text{let } \text{box } u = M \text{ in } \text{box } u \]
Summary of Reductions

- Reductions as basis for operational semantics.

  \((\lambda x:A. M) N \rightsquigarrow_R [N/x]M\)

- **let** box \(u = \text{box} M\) **in** \(N \rightsquigarrow_R \lbrack M/u \rbrack N\)

- Expansions as extensionality principles.

  \(M : A \rightarrow B \rightsquigarrow_E (\lambda x:A. M x)\)

  \(M : \Box A \rightsquigarrow_E (\text{let box} u = M \text{ in box} u)\).
Some Examples

- Application
  
  \( \vdash \lambda x:\Box(A \rightarrow B). \lambda y:\Box A. \)
  
  \[
  \text{let } \text{box } u = x \text{ in let } \text{box } w = y \text{ in box } (u \ w)
  \]
  
  : \Box(A \rightarrow B) \rightarrow \Box A \rightarrow \Box B

- Evaluation
  
  \( \vdash \lambda x:\Box A. \text{let } \text{box } u = x \text{ in } u \)
  
  : \Box A \rightarrow A

- Quotation
  
  \( \vdash \lambda x:\Box A. \text{let } \text{box } u = x \text{ in box } (\text{box } u) \)
  
  : \Box A \rightarrow \Box \Box A
Logical Assessment

- □ satisfies laws of intuitionistic S₄.

- Cleaner and simpler formulation through judgmental reconstruction.

- Can be extended to capture ◇.

- (An aside: model Moggi’s computational meta-language

\[ □A \quad \text{Value of type } A \]
\[ ◇□A \quad \text{Computation of type } A \]
\[ ◇□A \quad = \bigcirc A \text{ of lax logic} \] )
Operational Semantics

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

- Rules

  $$\mathbf{box} M \rightarrow \mathbf{box} M$$

  $$M \rightarrow \mathbf{box} M' \quad [M'/u] N \rightarrow V$$

  $$(\text{let } \mathbf{box} u = M \text{ in } N) \rightarrow V$$

- $\mathbf{box} M$ 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

  \[
  \text{let box } \boxed{u} = M \text{ in } \ldots u \ldots
  \]

  where \( u \) is not inside a \textbf{box} constructor.

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

- A \textbf{case} construct for source expressions(!) which does not violate \( \alpha \)-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 \( \text{box}(2^{2^{2^{2^2}}} - 1) \) 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 \texttt{box }M \text{ 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 $\Diamond A$ (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 $\text{box } M$ where $M$ is closed.

- Compile $M$ obtaining binary $B$ (using ML-RISC).

- Write code $C$ to generate $B$.

- Generate binary for $\text{box } M$ 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:

  \[
  \text{box } M \mapsto \lambda x : \text{unit}. M
  \]

  \[
  \text{let box } u = M \text{ in } N \mapsto \text{let val } x = M \text{ in } [x() / u]N
  \]

- Observationally equivalent, but prohibits any optimizations.
Invoking Generated Code

- Compiling `let box u = N in ... u ..., 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  \iff s = s_1 \circ s_2 \text{ where } s_1 \in \mathcal{L}(r) \text{ and } k s_2  \rightarrow \text{true} \text{ for some } s_1 \text{ 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
  in
    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

\[
\begin{align*}
| \text{acc (Const}(c)) &= \\
\text{let } \text{box } c' &= \text{lift } c \quad (* c : \text{string } *) \\
\text{in} \\
\text{box (fn } k => (\text{fn } (x::s) => (x = c') \text{ andalso } k \hspace{1em} s \\
| \text{nil } => \text{false})) \\
\text{end} \\
(* \text{val accept3 : regexp } -> (\text{string list } -> \text{bool}) *) \\
\text{fun accept3 } r = \\
\text{let } \text{box } a &= \text{acc } r \\
\text{in} \\
a \text{List.null} \\
\text{end}
\end{align*}
\]
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.

\[
\begin{align*}
\text{box} & \ (fn\ k \Rightarrow \ fn\ s \Rightarrow \\
& \quad (fn\ k \Rightarrow (fn\ (x::s) \Rightarrow (x = "a")\ \text{andalso}\ k \ s \\
& \quad \quad | \ \text{nil} \Rightarrow \ false)) \\
& \quad (fn\ ss \Rightarrow (fn\ k \Rightarrow fn\ s \Rightarrow k\ s)\ k\ ss)\ s) \\
\Rightarrow & \\
\text{box} & \ (fn\ k \Rightarrow fn\ s \Rightarrow \\
& \quad (fn\ (x::s') \Rightarrow (x = "a")\ \text{andalso} \\
& \quad \quad (fn\ ss \Rightarrow (fn\ k \Rightarrow fn\ s \Rightarrow k\ s)\ k\ ss)\ s' \\
& \quad \quad | \ \text{nil} \Rightarrow false))\ s) \\
\Rightarrow & \\
\text{box} & \ (fn\ k \Rightarrow fn\ s \Rightarrow \\
& \quad (fn\ (x::s') \Rightarrow (x = "a")\ \text{andalso} k\ s' \\
& \quad \quad | \ \text{nil} \Rightarrow false))\ s)
\end{align*}
\]

43
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.

```haskell
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.