Sequent Calculus as a Compiler Intermediate Language

Paul Downen [1] Luke Maurer [1]
Zena M. Ariola [1] Simon Peyton Jones [2]

[1]University of Oregon

[2]Microsoft Research Cambridge

ICFP'16, September 18 – 28, 2016

Curry-Howard in theory and practice

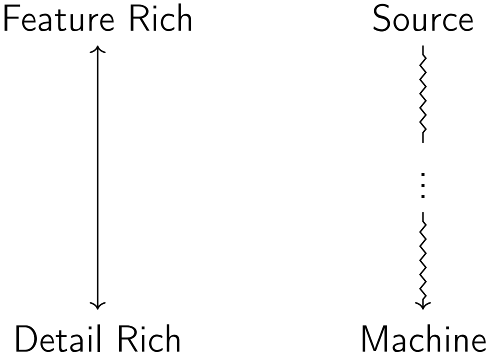
- Functional programming: wonderful marriage between theory and practice
- λ-calculus and natural deduction not just theoretical; a practical toolset for the real world
- Great basis for programming languages
- But what about *intermediate languages* in compilers?

Sequent Calculus as an Intermediate Language

- λ-calculus has been used in compilers for decades
- But λ's not the only game in town; the *sequent calculus* is another useful intermediate language
  - Low-level representations (Ohori, 1999a)
  - A logic (Ohori, 1999b) for *administrative-normal forms* (Flanagan et al., 1993)
  - Memory management via *structural rules* (Ohori, 2003)
  - Intuitionistic restrictions for functional purity
- A sequent-based language fits between λ-calculus and *continuation-passing style*

Intermediate Languages

A Compiler's Job



Feature Rich

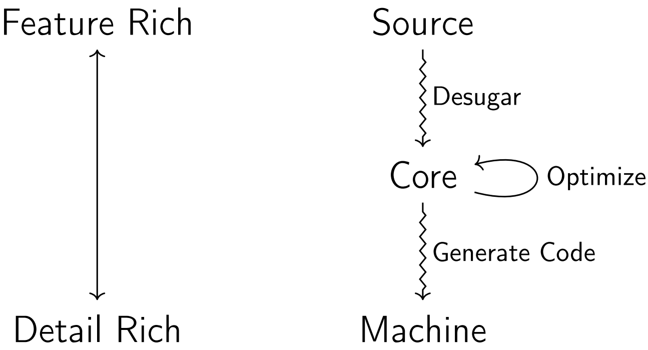
Detail Rich

Source

Machine

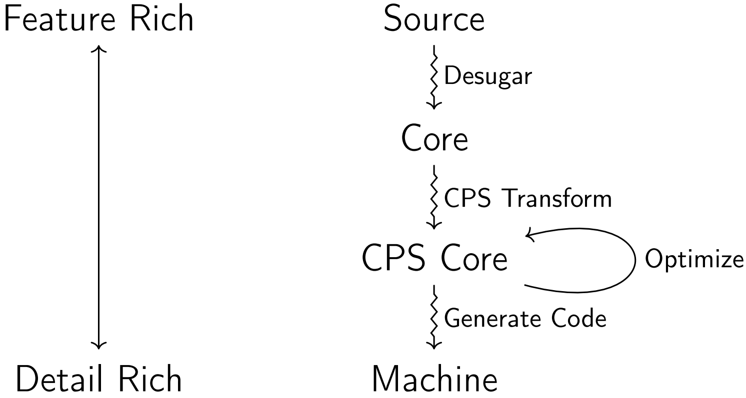
But this is a big jump; what goes in the middle?

Direct-Style IL



- Optimizations account for evaluation strategy
- Core = λ-calculus + polymorphism + data types

Continuation-Passing-Style IL



Feature Rich

Detail Rich

Source

Desugar

Core

CPS Transform

CPS Core

Optimize

Generate Code

Machine

- CPS transform bakes in evaluation strategy
- CPS Core = Core - non-tail-calls

The Sequent Calculus

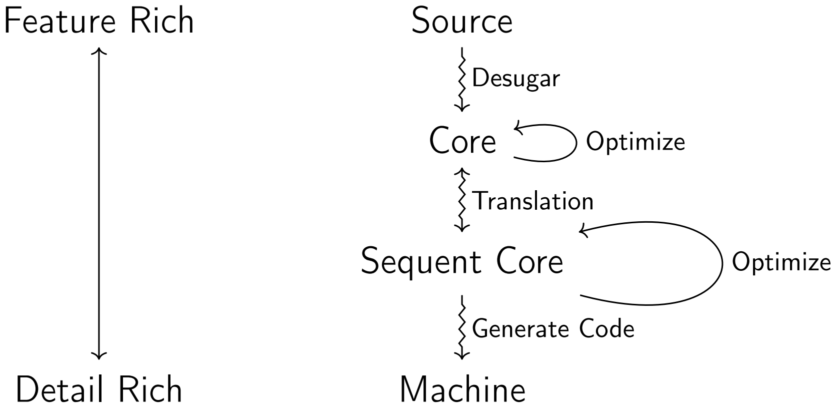
Gentzen's Two Logics

- Natural Deduction: "closer to mathematician's reasoning"
- Sequent Calculus: "easier to reason about"
- Natural Deduction $\approx$ λ-calculus
- Sequent Calculus $\approx$???

An (Abstract) Abstract Machine Language

- Language with left-right dichotomy: producers (values v) and consumers (continuations k) (Curien and Herbelin, 2000)
- Primary composition (a cut $\langle v \| k \rangle$) resembles an abstract machine state
- Still has high-level features: binding, substitution
- *Gentzen discovered statically-typed call-stacks in the 1930s*

Sequent-Style Intermediate Language



- Two-Way translation doesn't care about evaluation strategy
- Sequent Core = sequent calculus counterpart to Core

(Natural) Core vs Sequent Core

- Core is a data-flow language
  - Everything about expressions that return values
- Sequent Core contrasts data-flow and control-flow
  - Results given by values
  - Continuations do things with results
  - Both can be given a name
  - Computation happens when the two meet
- Two-way translation preserves semantics and types: can have best of both worlds!

The Two Roles of Continuations

Continuations as Evaluation Contexts

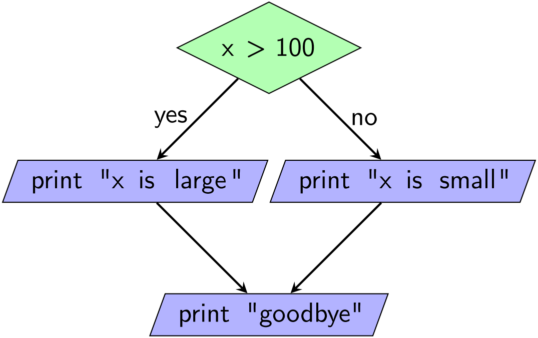
$(f(0) + 1) \times 2$

Take f;
Apply it to 0;
Add 1;
Multiply by 2;

- Say what to do with the intermediate results in a program
- Evaluation contexts are about *doing*

Continuations as Join Points

if $x > 100$:
  print `"x is large"`
else :
  print `"x is small"`
print `"goodbye"`



- A common point where several branches of control flow join together (ϕ node in SSA)
- Join points are about *sharing*

Evaluation Contexts vs Join Points

- The two are different in pure, lazy languages
- Evaluation contexts:
  - Take exactly one input
  - Are strict in their input
  - Cannot be run more than once
  - Can be scrutinized (use rewrite rules matching "call patterns")
- Join points:
  - Take zero or more inputs
  - May not need their input
  - Can be run many times (via recursion)
  - Are inscrutable (like a λ-abstraction)

Functions vs Join Points

- “But ‘join points’ sound a lot like functions!”
- They are, but very special functions:
  - Always tail-called, don’t return
  - Never escape their scope
- Different operational reading: just a jump to a labeled block of code
- Join points are more efficient to implement, less costly than a full closure

Sequent Core in GHC

Implementation

- Sequent Core implemented as a GHC plugin (`http://github.com/lukemaurer/sequent-core`)
- Use two-way translation to lift Sequent Core optimizations into Core-to-Core passes
- Implemented analogues of GHC optimizations/analyses on Sequent Core (The Mighty Simplifier, Let Floating, . . .)
- Found Sequent Core is better at *join points*

Case-of-Case and Friends

In Core:

$$\begin{array}{l} \textbf{let}\, j\ x\ y = \mathit{big} \\ \textbf{in}\, \mathit{not} (\textbf{case}\, z\, \textbf{of}\, \begin{array}{ll} A\ x\ y & \to j\ x\ y \\ B & \to \mathit{False}) \end{array} \end{array}$$

$$\Downarrow$$

$$\begin{array}{l} \textbf{let}\, j\ x\ y = \mathit{big} \\ \textbf{in}\, \textbf{case}\, z\, \textbf{of}\, \begin{array}{ll} A\ x\ y & \to \mathit{not}\ (j\ x\ y) \\ B & \to \mathit{not}\ \mathit{False} \end{array} \end{array}$$

This is bad! The join point is ruined (*j* no longer tail-called)

Case-of-Case and Friends

In Sequent Core (using Core syntax):

$$\begin{aligned}&\textbf{let}\, j\ x\ y = big\\&\textbf{in}\ \color{red}{not}\color{black}(\textbf{case}\, z\, \textbf{of}\ A\ x\ y \to j\ x\ y\\&\qquad\qquad\qquad\quad B \qquad\ \to False)\end{aligned}$$

$$\Downarrow$$

$$\begin{aligned}&\textbf{let}\, j\ x\ y = \color{red}{not}\color{black}\ big\\&\textbf{in}\, \textbf{case}\, z\, \textbf{of}\ A\ x\ y \to j\ x\ y\\&\qquad\qquad\quad\ B \qquad\ \to \color{red}{not}\color{black}\ False\end{aligned}$$

This is much better! The join point is preserved!

(Re-)Contification

- Sequent Core robustly preserves this status through optimizations (Yay!)
- But Core does not "know" about join points; they're lost in translation (Boo!)
- Contification: find functions that "look like" join points, and make them join points (Fluet and Weeks, 2001)
- Re-Contification (remembering lost join points after translation) is essential to the pipeline

Evaluation

- Benchmarks of Sequent Core optimizations competitive with Core
  - Similar performance, with occasional wins and losses
- Biggest cause for change (esp. losses): *inlining*
  - Inlining heuristics are tuned for Core; both very subtle and driving force for optimizations
  - With such a drastic change, can't pinpoint a root cause
- Modifying Core and original Simplifier would give clearer view on the impact of join points
- Need to pursue further optimizations for *cascading* effects

More in the paper

- Thorough description of the static and dynamic semantics of Sequent Core:
    - Type system
    - Call-by-name operational semantics: for reasoning about results
    - Call-by-need abstract machine: for operational reading of join points
- Purity via static scope restriction (Kennedy, 2007)
- Translations to and from Core
- Lightweight contification algorithm for translation

What Did Sequent Core Teach Us?

- “Continuations” serve (at least) two roles
- Sequent calculus is great at representing *negative types* (functions)
  - As GHC’s Might Simplifier already knew!
- Not just intuitionistic: join points are classical feature that can be tamed for purity
- Go beyond administrative-normal form
- Control flow not just for strict languages; it’s great for lazy languages, too

What Do We Want in an Intermediate Language?

	Direct	Sequent	CPS
Simple grammar	+		
Operational reading	+	++	++
Flexible eval order	+	+	−
Control flow	−	++	++
Rewrite rules	+	+	−

Current and Future Work

- From Sequent Core, extend Core with direct-style join points
- Improve optimizations (like contification) by inducing cascading effects
- Use Sequent Core as a laboratory for more context-aware opportunities using control flow

References I

P. Curien and H. Herbelin. The duality of computation. In *ICFP*, 2000. doi: 10.1145/351240.351262.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI*, 1993. doi: 10.1145/155090.155113.

M. Fluet and S. Weeks. Contification using dominators. In *ICFP*, 2001. doi: 10.1145/507635.507639.

A. Kennedy. Compiling with continuations, continued. In *ICFP*, 2007. doi: 10.1145/1291151.1291179.

References II

A. Ohori. The logical abstract machine: A curry-howard isomorphism for machine code. In *FLOPS*, 1999a. doi: 10.1007/10705424_20.

A. Ohori. A curry-howard isomorphism for compilation and program execution. In *TLCA*, 1999b. doi: 10.1007/3-540-48959-2_20.

A. Ohori. Register allocation by proof transformation. In *ESOP*, 2003. doi: 10.1007/3-540-36575-3_27.