Confluences in Programming Languages Research

David Walker
Princeton University
A confluence: The junction of two rivers
A Confluence of Ideas

Stream I

Stream II

Inflection Point:
A change in the world.

A Confluence of Ideas

Main Stream

Stream I

Stream II

Impacting Stream

Inflection Point

Transformed Ideas

New Environment = New Rules, New Opportunities

Impact

New Stream

Impressionism as Confluence

Invisible strokes and a focus on realistic detail

Realism (1800s)

Photography
“Press a button, we do the rest” (Kodak 1888)

Impressionism as Confluence

Invisible strokes and a focus on realistic detail

Realism (1800s)

Psychology

Freud (1836-1934): emotion affects perception

Photography “Press a button, we do the rest” (Kodak 1888)

Impressionism as Confluence

Invisible strokes and a focus on realistic detail

Visible strokes, emotion and movement

Realism (1800s)

Psychology

Freud (1836-1934): emotion affects perception

Photography “Press a button, we do the rest” (Kodak 1888)

Networked Vehicles as Confluence

Threat model: “watch out for that horse”

Motorized Vehicles
Networked Vehicles as Confluence

Motorized Vehicles

cars go online

Threat model: “watch out for that horse”
Networked Vehicles as Confluence

Threat model: “watch out for that horse”

New threat models: Hackers remotely take control of Jeep on highway

Motorized Vehicles

cars go online
Networked Vehicles as Confluence

Threat model: “watch out for that horse”

Motorized Vehicles

Formal methods

cars go online

New threat models: Hackers remotely take control of Jeep on highway
Networked Vehicles as Confluence

Threat model: “watch out for that horse”

New threat models: Hackers remotely take control of Jeep on highway

Motorized Vehicles

Formal methods

cars go online

High Assurance Vehicular Systems
Networked Vehicles as Confluence

Threat model: “watch out for that horse”

New threat models: Hackers remotely take control of Jeep on highway

- Motorized Vehicles
- Application Domain
- Formal methods
- Basic PL Research
- High Assurance Vehicular Systems
- cars go online
Operating System Reliability as Confluence

Basic research breakthroughs:
- data structures
- algorithms
- abstraction

Hardware:
- thanks Intel!
Operating System Reliability as Confluence

Blue Screen of Death

Operating System Reliability

Model Checking

Static Driver Verification

Basic research breakthroughs:
- data structures
- algorithms
- abstraction

Hardware:
- thanks Intel!
Confluences in Programming Languages Research

- Inflection Point
- Application Domain
- Basic PL Research

Application Impact

- Hardware Changes
- Technology Changes
- Scale Changes
- Complexity Changes
- Economy Changes

Basic Research Breakthroughs:
New Techniques, Tools and Algorithms
Why Confluences?

Inflection points separate fads from opportunity for real change.

Early access maximizes influence on thought leaders.
How Confluences?

Can we really see these inflection points as they happen?

Not always! **We (often) can’t!**

- **Application Domain**
  - We can make friends.
  - Perhaps *they* can

- **Basic PL Research**
  - Deep, general, reusable, hard-to-learn skills

- **Change the world with collaborative research**
  - **Application Impact**
Two Confluences In My Career
And What I Learned From Them

Grad School: Learning Skills
• Confluences in reliable systems implementation
• Inflection point: a breakthrough in basic research

Professor Life: Making Friends
• Confluences in network configuration
• Inflection point: growth of data centers & industrial networks
Grad School: Learning Skills
Confluences in Reliable Systems Implementation

With Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Fred Smith, Stephanie Weirich, Steve Zdancewic
Stream 1: Basic Research: Type Safety
An Ever-So-Brief History of Modern Type Safety Proofs

Dynamic Typing in a Statically Typed Language

Abstract

Statically typed programming languages allow earlier error checking, better enforcement of disciplined programming styles, and generation of more efficient object code than languages where all type consistency checks are performed at run time. However, even in statically typed languages, there is often the need to deal with data whose type cannot be determined at compile time. To handle such situations, we propose to add a type Dynamic whose values are pairs of a value v and a type tag T where the type denoted by T is one of several instances of Dynamic. We also propose a type-safe Dynamic constraint.

This paper explores the syntax, operational semantics, and denotational semantics of a simple language including the type Dynamic. We give examples of how dynamically typed values can be used in programs. Then we discuss an operational semantics for our language and obtain a soundness theorem. We present an implementation of the denotational semantics of this language and relate them to the operational semantics. Finally, we consider the implications of polymorphism and some implementation issues.

Dynamic Typing in a Statically Typed Language. Abadi, Cardelli, Pierce, Plotkin.

Semantic Domains:

- \( V = B_0 + B_1 + \ldots + F + W + D \)
- \( F = V \rightarrow V \)
- \( D = \text{TypeCode} \times V \)
- \( W = \{ . \} \)

Proof:

Metric space argument shows the existence of the semantic relation.

History from: A Syntactic Approach to Type Soundness. Wright and Felleisen. 1994
Where we were at:

Real systems, Languages

gap

Type Safety

Milner  Tofte  MacQueen
Damas  Mitchell
Plotkin  Harper  Felleisen
Martin-Lof  Kahn  Lillibridge
Duba  Talpin  Gifford
Friedman  … Many More …

Tiny, elegant languages

Hard proofs that are getting harder and that change with each new feature
The Inflection Point: Simple Syntactic Methods

A Syntactic Approach to Type Soundness

Andrew K. Wright and Matthias Felleisen

Department of Computer Science, Rice University, Houston, Texas 77251-1892

We present a new approach to proving type soundness for Hindley-Milner-style polymorphic type systems. The keys to our approach are (1) an adaptation of subject reduction theorems from combinatory logic to programming languages, and (2) the use of rewriting techniques for the specification of the language semantics. The approach easily extends from polymorphic functional languages to imperative languages that provide references, exceptions, continuations, and similar features. We illustrate the technique with a type soundness theorem for the core of Standard ML, which includes the first type soundness proof for polymorphic exceptions and continuations.

Key contributions:

- Semantics by syntactic program rewriting
- Check program states are well-typed at each step
- Modern Type Preservation
- Demonstrated reuse of the same technique on a variety of features and series of languages

A Simplified Account of Polymorphic References


A Simplified Account of Polymorphic References

Robert Harper

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

A simplified account of the semantics of polymorphic types for the call-by-name operational semantics in simply-typed lambda calculus. The presentation in this paper is based on the formalism of the operational semantics of Harper and Mitchell briefly. The focus being a particular simple proof of soundness of Finley's type discipline.

Modern Canonical Forms, Progress!

Harper, influenced by Martin-Löf, Plotkin
Confluences in Reliable Systems Implementation

Type safety in practice:

- the foundation of mobile code security (Java & JVM)
- the foundation of promising systems architectures (SPIN OS)
- typed interfaces + type safety = secure, efficient sandboxes

But type checking happened at the source

- consumers had to trust a compiler to preserve safety invariants
- compilers are 100s of thousands, millions LOC — errors inevitable

Can we pull the compiler out of the trusted computing base?
Typed Intermediate and Target Languages

From System F to Typed Assembly Language. Morrisett, Walker, Crary, Glew. POPL 98.
% sum: eax + 1 + 2 + … + ebx
% eax: accumulator
% ebx: counter
% ecx: continue with result in eax

sum:

beq ebx, ecx  % if ebx=0, jump to ecx
add eax, ebx  % eax := eax + ebx
sub ebx, 1    % decrement counter
jump sum      % iterate loop

Register files R:
R ::= {eax = v, ebx = v, …}

Register file types Γ:
Γ ::= {eax : τ, ebx : τ, …}

Machine value types:
τ ::= int32
    | int64
    | float32
    | Γ    % code ptr
    | α    % abstract type
    | ∀α.τ % universal
% sum: eax + 1 + 2 + … + ebx
%
% eax: accumulator
% ebx: counter
% ecx: continue with result in eax

sum:

beq ebx, ecx  % if ebx=0, jump to ecx

add eax, ebx  % eax := eax + ebx

sub ebx, l  % decrement counter

jump sum  % iterate loop

sum’s code type:
{
  eax: int32,
  ebx: int32,
  ecx: {eax: int32}
}
% sum: eax + 1 + 2 + ... + ebx
%
% eax: accumulator
% ebx: counter
% ecx: continue with result in eax

sum:

{ eax: int32, ebx: int32, ecx : {eax: int32} }

beq ebx, ecx  % if ebx=0, jump to ecx

add eax, ebx  % eax := eax + ebx

sub ebx, 1  % decrement counter

jump sum  % iterate loop
TALx86

% sum: eax + 1 + 2 + … + ebx
%
% eax: accumulator
% ebx: counter
% ecx: continue with result in eax

sum:

{ eax: int32, ebx: int32, ecx :{eax: int32} }

beq ebx, ecx  % if ebx=0, jump to ecx

{ eax: int32, ebx: int32, ecx :{eax: int32} }

add eax, ebx  % eax := eax + ebx

sub ebx, 1     % decrement counter

jump sum       % iterate loop
% sum: eax + 1 + 2 + … + ebx
% % eax: accumulator
% % ebx: counter
% % ecx: continue with result in eax

sum:

{ eax: int32, ebx: int32, ecx : {eax: int32} }

beq ebx, ecx  % if ebx=0, jump to ecx
{ eax: int32, ebx: int32, ecx : {eax: int32} }

add eax, ebx  % eax := eax + ebx
{ eax: int32, ebx: int32, ecx : {eax: int32} }

sub ebx, 1  % decrement counter
{ eax: int32, ebx: int32, ecx : {eax: int32} }

jump sum  % iterate loop

sum’s code type:
{
  eax: int32,
  ebx: int32,
  ecx: {eax: int32}
}
Modelling Calling Conventions

% sum: eax + 1 + 2 + … + ebx
%
% eax: accumulator
% ebx: counter
% ecx: continue with result in eax

sum’s code type:
{
  eax: int32,
  ebx: int32,
  ecx: {eax: int32}
}

a different calling convention:
∀α.{
  eax: int32,
  ebx: int32,
  ecx: {eax: int32, edx: α},
  edx: α
}

Callee (sum) saves register:
Type abstraction requires the callee to act parametrically in α

Extensions: Stack Typing

% sum: esp[0] + 1 + 2 + … + esp[4]
% esp[0]: accumulator
% esp[4]: counter
% esp[8]: continue with result in eax
Extensions: Stack Typing

% sum: esp[0] + 1 + 2 + … + esp[4]
% esp[0]: accumulator
% esp[4]: counter
% esp[8]: continue with result in eax

modelling stacks s as lists:

s ::= nil | v :: s

stack types σ via an algebra of lists:

σ ::= nil % empty stack
| τ :: σ % a value on top
| ρ % an abstract stack
| σ @ σ % two stack segments
Extensions: Stack Typing

% sum: esp[0] + 1 + 2 + … + esp[4]
% esp[0]: accumulator
% esp[4]: counter
% esp[8]: continue with result in eax

stack-based sum’s code type:
∀ρ. {  
  esp: int32 ::  
  int32 ::  
  {eax: int32, esp: ρ} ::  
  ρ,
}

esp:
accumulator
counter
call site
ρ
Extensions: Stack Typing

% sum: esp[0] + 1 + 2 + … + esp[4]
% esp[0]: accumulator
% esp[4]: counter
% esp[8]: continue with result in eax

Parametric polymorphism prevents the callee from trampling on the caller’s stack

∀ρ.{
    esp: int32 ::
    int32 ::
    {eax: int32, esp: ρ} ::
    ρ,
}

TALx86 Summary

More types:

• For closures, data types, arrays, exceptions
• Types and kinds for describing object sizes, memory allocation and initialization
• Linking
• …

Moral of the story:

• Basic research in types reused in an extreme new setting
  • Impossible without syntactic proof techniques
• Biggest contribution:
  • Showing fully automatic proof of strong safety properties in general-purpose assembly is possible
Non-technical Take-aways

Learn a small number of highly reusable skills really well.

- I learned one non-trivial proof technique:
  - Progress and Preservation
  - I practiced it over and over

- I learned how to develop small models:
  - idealized operational models with abstract objects
    - stacks, heaps, registers, …
  - tiny type systems, simple algebras
    - simplicity takes practice and experience
      - nobody ever uses or remembers the complicated stuff

I have used almost nothing else for the rest of my career.

(Perhaps I’m lazy)
The Confluence Continues

- System Reliability
- Safe Language Theory
- Logical Frameworks
- Testing
  - Syntactic Methods
- Type Checking
- Improved Tools
  - Github, Open Source Theorems
- Safer Systems
- Unbreakable Systems
  - Compcert
  - seL4
  - CertiKOS
  - RockSalt
  - DeepSpec
- Theorem Proving
Professor Life: Making Friends
Confluences in Network Configuration

With Carolyn Jane Anderson, Ryan Beckett, Nate Foster, Michael Greenberg, Arjun Guha, Stephen Gutz, Rob Harrison, Jean-Baptiste Jeannin, Naga Praveen Katta, Dexter Kozen, Mathew Meola, Chris Monsanto, Josh Reich, Mark Reitblatt, Jennifer Rexford, Cole Schlesinger, Alec Story, Todd Warszawski
Network Configuration
Traditional Networks

Each router:

- maintains its own view of the world
- uses a standard protocol to communicate with neighbours and select routes

Network operators select from these standard, pre-defined protocols

- Operators supply parameters to configure them

Hardware vendors (eg, CISCO) control the software

- Protocol standards evolve slowly

```plaintext
ospf interface ip metric 3
ospf ...
ospf ...
ospf-passive ...
... ip 10.0.0.0/24
ospf redistribute metric 10
bgp ... x ... C apply ...
```
The Inflection Point

Technological & Economic Changes:

Data center infrastructure scales up

Owners of this infrastructure stand to gain from customized and centralized network control algorithms.
Connecting Inter-continental Cloud Services

Traditional WANs:
- No control over end hosts
- All bits treated the same
- 30-40% utilization achieved
  - overprovisioning for fault tolerance

B4 WAN Connects Google’s Data Centers:
- Control over end applications — limit their sending rate
- Multiple traffic classes, treated differently
  - user traffic: low volume, latency sensitive
  - big data synchronization: high volume, latency insensitive, fault tolerant
- Through centralized route control and traffic engineering, link utilization nears 100% on some links. Averages 70% or more throughout. 2x-3x cost savings.
Software-Defined Networking (SDN): The Technology Behind B4

Centralized, General-purpose Controller Machine

OpenFlow

Centralized controller plans routes using global information

- Rather than configuring distributed algorithms, the controller tells each switch how to forward, modify or drop packets directly

- OpenFlow: The new “network assembly language”
  - simple
  - yet expressive, capable of constructing any path
Confluences in Network Configuration

- Economic Changes:
  - Network Infrastructure Growth
  - Data Centers

- Technology Change:
  - OpenFlow

- Distributed Protocols
- Network Configuration

- Centralized Control
Confluences in Network Configuration

Distributed Protocols

Network Configuration

Modular programming and reasoning

Economic Changes:
Network Infrastructure Growth
Data Centers

Technology Change:
OpenFlow

Centralized Control

Modular Network Programming Languages
Software-Defined Network (SDN) Programming

Event

Install R₁ on A
Install R₂ on A
Install R₃ on B
Install R₄ on A
Remove R₂ from A
...

A

B
SDN Programming

Event

Install R₁ on A
Install R₂ on A
Install R₃ on B
Install R₄ on A
Remove R₂ from A

High variability in reaction time: seconds or even minutes
SDN Programming

Event

Install R₁ on A
Install R₂ on A
Install R₃ on B
Install R₄ on A
Remove R₂ from A

High variability in reaction time: seconds or even minutes

At the same time, switch continues processing incoming packets at line rate
Early SDN: Event-driven, imperative, concurrent programming with distributed, stateful tables read asynchronously by other agents
Frenetic: Structured SDN Programming

Event

Compositional, Global, Declarative Policy

Compute

A

B

C

Frenetic: A Network Programming Language. Foster, Harrison, Freedman, Monsanto, Rexford, Story, Walker. ICFP 2011
Frenetic: Structured SDN Programming

Frenetic: A Network Programming Language. Foster, Harrison, Freedman, Monsanto, Rexford, Story, Walker. ICFP 2011
Programmer’s View
Programmer’s View

... Event 2

Event 1
Programmer’s View

We need some protocol for updating switches.

If we aren’t careful a lot of bad stuff could happen:
  • packets from X to Y could be dropped
  • packets could be mis-directed
  • congestion?

Clearly, the protocol should preserve some “good” properties across updates

Underlying Physical Network
Preserving Properties

What kinds of properties?

- *Per-packet Path Properties (PPP)*: Any property of a single packet, its path through the network, and modifications along the way
  - Access control
  - Reachability
  - Way-pointing
  - But not congestion (a property of many packets)

Which ones?

- *All of them*: Preserve any PPP shared by 2 consecutive policies
- *Advantage*: Programmers don’t need to supply invariants
  - *Advantage*: To check Inv is preserved forever, check all policies independently

How?

- *Per-packet Consistent Update*: Ensure every packet traverses either the old policy or the new policy, not some mixture of both
 Implementation Mechanism: 2-phase Commit

Preprocess every policy:
- Entry locations stamp policy version number on packets (green/blue)
- Internal location apply their policy if the packet carries the right number

To update from green to blue:
- **Phase 1**: Add new blue rules to internal switches, while packets continue to be stamped green and are processed by green rules
- **Phase 2**: Overwrite entry location green-stamping rules with blue-stamping rules

Improvements and Refinements

Incremental updates trade time for space [Katta et al., HotSDN 2013]

Updates with congestion control [Hong et al., SIGCOMM 2013]

Dynamic update scheduling improves update time [Jin et al., SIGCOMM 2014]

Preserving user-supplied invariants instead of all invariants improves update time and space! [McClurg, PLDI 2015]
Consistent Updates: Modular Reasoning in Time

... Event 2

Event 1

Frenetic Policy Lang's: Modular Reasoning in Space

Frenetic [ICFP 11], NetCORE [POPL 12], Pyretic [NSDI 13], NetKAT [POPL 14], SDX [SIGCOMM 14], Fast NetKAT [ICFP 15], Concurrent NetCORE [ICFP 15], CoVisor [NSDI 15], Kinetic [NSDI 15], Probabilistic NetKAT {ESOP 16}, Path Queries [NSDI 16] ...
Technical Take-aways

The networking community has embraced language-based approaches to network configuration.

ACM Symposium on SDN Research (SOSR) sponsored by SIGCOMM topics include:
- Programming languages, verification techniques and testing techniques for SDN

P4: A Language-based “OpenFlow 2.0”
- start: a PL/networking group [SIGCOMM CCR 2014]
- now: 33 member organizations (as of Dec 14, 2015)
- several PL folks providing feedback

MOOC: Software-Defined Networking
- Nick Feamster (Georgia Tech → Princeton)
- 870 students doing assignments, survey
- 217 full-time network operators
- 79% preferred Kinetic [NSDI 15] to current approaches
- 84% agreed it helped make it easier to verify policies
Non-Technical Take-aways

Sometimes research is all about the detailed result:

• Progress and Preservation

But sometimes it is people and communication that matter most:

• We got in to SDN early when no one had written any programs. How?
  • Our colleague Jen Rexford was at the forefront of the area
    • She developed the intellectual precursors to SDN at AT&T
    • She spotted the SDN inflection point
    • She was open-minded
  • We wrote a grant together
  • We had beyond-brilliant colleagues (Nate Foster and others)

• Then we got mind share. How?
  • Jen gave early an keynote talk at the Open Networking Summit
  • Followed up the next year by Nate Foster
  • Jen gave many, many industrial talks; she has many friends

Moral: Make Friends
Summary:
Confluences in Programming Language Research

Make friends

Learn re-useable PL skills

Be open-minded
Watch for the inflection points

Application Domain

Basic Research

Application Impact

Transform the world
thank you