What Program Analysis Can and Cannot Do for You

David Van Horn
with support from NSF, CRA, Google.
A Formulae-as-Types Notion of Control

Timothy G. Griffin
Department of Computer Science
Rice University
Houston, TX 77251-1892

Abstract

The programming language Scheme contains the control construct call/cc that allows access to the current continuation (the current control context). This, in effect, provides Scheme with first-class labels and jumps. We show that the well-known formulae-as-types correspondence, which relates a constructive proof of a formula $\alpha$ to a program of type $\alpha$, can be extended to a typed Idealized Scheme. What is surprising about this correspondence is that it relates classical proofs to typed programs. The existence of computationally interesting “classical programs” — programs of type $\alpha$, where $\alpha$ holds classically, but not constructively — is illustrated by the definition of conjunctive, disjunctive, and existential types using standard classical definitions. We also prove that in general, classical proofs lack computational content. This paper shows, however, that the formulae-as-types correspondence can be extended to classical logic in a computationally interesting way. It is shown that classical proofs possess computational content when the notion of computation is extended to include explicit access to the current control context.

This notion of computation is found in the programming language Scheme [16], which contains the control construct call/cc that provides access to the current continuation (the current control context). This, in effect, provides Scheme with first-class labels and jumps, and allows for programs that are more efficient than purely functional programs. The formulae-as-types correspondence presented in this paper is based on a typed version of Idealized Scheme — a typed ISWIM containing an operator
classical proofs to typed programs. The existence of computationally interesting "classical programs" — programs of type \( \alpha \), where \( \alpha \) holds classically, but not constructively — is illustrated by the definition of conjunctive, disjunctive, and existential types using standard classical definitions. We also prove that the current computation (the current control context). This, in effect, provides Scheme with first-class labels and jumps, and allows for programs that are more efficient than purely functional programs. The formulae-as-types correspondence presented in this paper is based on a typed version of Idealized Scheme — a typed ISWIM containing an operator...
classical proofs to typed programs. The existence of computationally interesting “classical programs” — programs of type $\alpha$, where $\alpha$ holds classically, but not constructively — is illustrated by the definition of conjunctive, disjunctive, and existential types using standard classical definitions. We also prove that

the current combination (the current control context). This, in effect, provides Scheme with first-class labels and jumps, and allows for programs that are more efficient than purely functional programs. The formulae-as-types correspondence presented in this paper is based on a typed version of Idealized Scheme — a typed ISWIM containing an operator
What Program Analysis Can and Cannot Do for You

David Van Horn
with support from NSF, CRA, Google.
Higher-order Program Analysis is Dead.

(I should know, I killed it.)
it’s hard to write
it’s slow
it’s imprecise
it’s awful
Higher-order Program Analysis is Alive and Well. (I have a way forward.)
it’s easy to write
it’s fast
it’s precise
it’s great
So what?
Modern software is higher-order.
We need reasonable software.
So you should care.
Modern software is higher-order.
Modern software is higher-order.

Q: What are higher-order languages?
Modern software is higher-order.

Q: What are higher-order languages?
A: Languages in which computations are values.
Modern software is higher-order.

```python
# (R → R) → (R → R)
def deriv(f):
    def fp(x):
        return ((f(x+e) - f(x-e)) / (2*e));
    return fp
```

```python
from math import sin
e = 0.0001
```

```python
print deriv(sin)(4)
```
Modern software is higher-order.

```
// (R → R) → (R → R)
public Func<Double, Double> deriv(final Func<Double, Double> f) {
    return new Func<Double, Double>() {
        public Double apply(Double x) {
            return ((f.apply(x+ε) - f.apply(x-ε)) / (2*ε));
        }
    };
}
```
Modern software is higher-order.

```csharp
// (R → R) → (R → R)
static Func<double, double>
deriv(Func<double, double> f) {
    return (x)
        => (f(x + ε) - f(x - ε)) / (2 * ε);
}
```
Modern software is higher-order.

```javascript
// (R → R) → (R → R)
function deriv(f) {
    return function (x) {
        return (f(x+ε) - f(x-ε)) / (2*ε);
    };
}
```

Modern software is higher-order.

// (R -> R) -> (R -> R)
def deriv(f: (Double) => Double) {
    val fp = (x: Double) =>
        (f(x+ε) - f(x-ε)) / (2*ε);
    return fp;
}

Modern software is higher-order.
3.5.3 Creating Functions Dynamically

We have already seen in 1.2.3 that when you build a function at run time you can take advantage of at least some of the identifiers visible to you in implementing the body of the function. Here’s the code we had there:

```
1 w val r = new Randommn;
2 x val rand = mn => rtnextDoublemn;
3 { val inCircle = countPointsmNr randn;
4 }
```

You’ll recall that this code sat inside a loop so every time through the loop executing line 2 gave us a new value of `r` to use with which to build a new function to assign to `rand`.

The question is: what identifiers from the surrounding context are allowed in the body of a function literal “or as it is more usually called) a “closure”? like `rand`?

The bad news is that you cannot use any of the method’s `local` identifiers that are declared to be `vars`. The good news is that anything else goes. Any identifier that is `val` or is `static` can be used. If the closure is being declared inside an instance method) then any instance member “field or method” of the class may appear in the literal’s body. This is okay because, even if `x` is a field that is a `var`) what is being captured by the closure is the value of `this` and `this` itself is a local `val` in the instance method: you cannot assign a value to `this` itself in the body of a method) you can only assign to the fields `this` references.

```
public class IntRange {
  val low: Int;
  var high: Int;
  public def this(low: Int, high: Int) {
    this.low = low; this.high = high;
  }
  public def includes(n:Int) = low <= n && n <= high;
  public static def isDigitFcn() {
    val digit = new IntRange(0,9);
    return (n: Int) => digit.includes(n);
  }
  public def inMeTester() {
    return (n: Int) => low <= n && n <= high;
  }
}
```
3.5.3 Creating Functions Dynamically

We have already seen in 1.2.3 that when you build a function at run-time) (you can take advantage of at least some of the identifiers visible to you in implementing the body of the function. Here's the code we had there:

```java
w val r = new Randomn;
x val rand = mn => rtnextDoublemn;
{ val inCircle = countPointsNr randn;
You'll recall that this code sat inside a loop), so every time through the loop, executing line 2 gave us a new value of r to use with which to build a new function to assign to rand.

The question is: what identifiers from the surrounding context are allowed in the body of a function literal "or as it is more usually called) a "closure"? like rand?

The bad news is that you cannot use any of the method's local identifiers that are declared to be vars. The good news is that anything else goes. Any identifier that is a val or is static can be used. If the closure is being declared inside an instance method) then any instance member (field or method? of the class may appear in the literal’s body. This is okay because) even if x is a field that is a var) what is being captured by the closure is the value of this) and this itself is a local val in the instance method: you cannot assign a value to this itself in the body of a method) you can only assign to the fields this references.

```java
public class IntRange {
  val low: Int;
  var high: Int;
  public def this(low: Int, high: Int) {
    this.low = low; this.high = high;
  }
  public def includes(n: Int) = low <= n && n <= high;
  public static def isDigitFcn() {
    val digit = new IntRange(0, 9);
    return (n: Int) => digit.includes(n);
  }
  public def inMeTester() {
    return (n: Int) => low <= n && n <= high;
  }
}
```
window.setTimeout

Summary

Executes a code snippet or a function after specified delay.

Syntax

```
var timeoutID = window.setTimeout(func, delay, [param1, param2, ...]);
var timeoutID = window.setTimeout(code, delay);
```

where

- `timeoutID` is the ID of the timeout, which can be used later with `window.clearTimeout`.
- `func` is the function you want to execute after `delay` milliseconds.
- `code` in the alternate syntax, is a string of code you want to execute after `delay` milliseconds. (Using this syntax is not recommended for the same reasons as using `eval()`)
- `delay` is the number of milliseconds (thousandths of a second) that the function call should be delayed by. Note that the actual delay may be longer, see Notes below.
window.setTimeout

Summary

Executes a code snippet or a function after specified delay.

Syntax

```javascript
var timeoutID = window.setTimeout(func, delay, [param1, param2, ...]);
var timeoutID = window.setTimeout(code, delay);
```

where

- `timeoutID` is the ID of the timeout, which can be used later with `window.clearTimeout`.
- `func` is the function you want to execute after `delay` milliseconds.
- `code` in the alternate syntax, is a string of code you want to execute after `delay` milliseconds. (Using this syntax is not recommended for the same reasons as using `eval()`)
- `delay` is the number of milliseconds (thousandths of a second) that the function call should be delayed by. Note that the actual delay may be longer, see Notes below.
Modern software is higher-order.

1. Introduction

This section is non-normative.

The XMLHttpRequest object implements and performs HTTP client functionality, such as ECMA-Script HTTP API.

The name of the object is XMLHttpRequest and name is potentially misleading. First, the object can be used to make requests over both HTTP and HTTPS, but that function "requests" in a broad sense of the term as requests or responses for the defined HTTP method.

Some simple code to do something with XMLHttpRequest:

```javascript
function test(data) {
  // taking care of data
}

function handler() {
  if(data.readyState == 4 && data.status == 200) {
    test(data.responseXML.getElementsByTagName('el1')[0].firstChild.textContent);
  } else if (data.readyState == 4 && data.status !!= 200) {
    test(null);
  } else if (data.readyState == 4 && data.status !== 200) {
    test(null);
  }
}

var client = new XMLHttpRequest();
client.onreadystatechange = handler;
client.open("GET", "unicorn.xml");
client.send();
```

Evented I/O for V8 JavaScript.

An example of a web server written in Node which responds with "Hello World" for every request.

```javascript
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World
');
}).listen(8124, "127.0.0.1");
console.log('Server running at http://127.0.0.1:8124/
');
```

To run the server, put the code into a file example.js and execute it with the node program:

```bash
% node example.js
Server running at http://127.0.0.1:8124/
```
public class Observable
extends Object

This class represents an observable object, or "data" in the model-view paradigm. It can be subclassed to represent an object that the application wants to have observed.

An observable object can have one or more observers. An observer may be any object that implements interface Observer. After an observable instance changes, an application calling the observable's notifyObservers method causes all of its observers to be notified of the change by a call to their update method.

The order in which notifications will be delivered is unspecified. The default implementation provided in the Observable class will notify Observers in the order in which they registered interest, but subclasses may change this order, use no guaranteed order, deliver notifications on separate threads, or may guarantee that their subclass follows this order, as they choose.

Note that this notification mechanism has nothing to do with threads and is completely separate from the wait and notify mechanism of class Object.

When an observable object is newly created, its set of observers is empty. Two observers are considered the same if and only if the equals method returns true for them.

Since:
JDK1.0

See Also:
notifyObservers(), notifyObservers(java.lang.Object), Observer,
Observer.update(java.util.Observable, java.lang.Object)
Modern software is higher-order.

### Constructor Summary

**Observable()**

Construct an Observable with zero Observers.

### Method Summary

- **addObserver(Observer o)**: Adds an observer to the set of observers for this object. The observer is already in the set.
- **clearChanged()**: Indicates that this object has no longer changed, and is the most recent change, so that the hasChanged method is set to true.
- **countObservers()**: Returns the number of observers of this Observable.
- **deleteObserver(Observer o)**: Deletes an observer from the set of observers of this Observable.
- **deleteObservers()**: Clears the observer list so that this object no longer has any observers.
- **hasChanged()**: Tests if this object has changed.
- **notifyObservers()**: If this object has changed, as indicated by the hasChanged method, then call the clearChanged method to indicate that this object has no longer changed.
- **notifyObservers(Object arg)**: If this object has changed, as indicated by the hasChanged method, then call the clearChanged method to indicate that this object has no longer changed.
- **setChanged()**: Marks this observable object as having been changed; the hasChanged method will return true.

---

**Library: observer**

The Observer pattern, also known as Publish/Subscribe, provides a simple mechanism for one object to inform a set of interested third-party objects when its state changes.

In the Ruby implementation, the notifying class mixes in the Observable module, which provides the methods for managing the associated observer objects.
Modern software is higher-order.

C++  Java  JavaScript
Python  Scheme
C#     X10 / Habanero  OCaml

...and many more
Modern software is higher-order.

...and many more

Higher order
Modern software is higher-order.

...and many more

Higher order
We need reasonable software.
We need reasonable software.

Q: What does it mean to reason about software?
We need reasonable software.

Q: What does it mean to reason about software?

A: It means predicting the future.
We need reasonable software.
We need reasonable software.
We need reasonable software.

```java
public void f(XYZ x) {
    x.m();
}
```

Optimizing Java compiler:
prove x is always an X, inline method definition.
We need reasonable software.

\texttt{first(x)}

Puzzled ML programmer: prove \( x \) is always a non-empty list: no problem.
We need reasonable software.

first(x)

Puzzled ML programmer: prove x is may be the empty list: fix.
We need reasonable software.

```
checkPrivilege(R);
```

Security analyzer: prove enable(R) is on the stack.
We need reasonable software.

Optimizing compilers  Parallelizing compilers  Software construction
Security analysis  Program understanding
Static debugging  Termination analysis  Model checking

...and many more
We need reasonable software.

Optimizing compilers   Parallelizing compilers   Software construction

Security analysis   Program understanding

Static debugging   Termination analysis   Model checking

...and many more

Program analysis
Higher-order program analysis

Scalability

- Complexity
- Maintenance
- Verification
- Expressivity
- Modularity
Q: What is program analysis?
Q: What is program analysis?

A: Prediction of which values show up at which program sites.
```csharp
using System;

public class Calculus {
    static double ε = 0.0001;

    // (x)!
    static Func<double, double> deriv(Func<double, double> f) {
        return (x) => (f(x + ε) - f(x - ε)) / (2 * ε);
    }

    static public void Main() {
        Console.WriteLine(deriv(Math.Sin)(4));
    }
}
```

Where does data go to?
Where does control go to?

\[ f(x + \varepsilon) \]
Who calls deriv?

def\text{deriv}\ (\text{Func<double, double>} \ f) \ { 
\text{return } \ (x) 
=> \ (f(x+\varepsilon) - f(x-\varepsilon)) \ / \ (2*\varepsilon) \ ; 
}

To do control-flow analysis, you need data-flow analysis  To do data-flow analysis, you need control-flow analysis
Why so tangled up?

Values include Computations

Computation is Code plus Data
Why so tangled up?

Values include Computations

Computation is Code plus Data
Why so tangled up?

Values include Computations

Computation is Code plus Data

Computable predictions about run-time behavior
Why so tangled up?

Values include Computations

Computation is Code plus Data

Computable predictions about run-time behavior

So what’s their complexity?
Existing analyses and their complexity
function app(f,x) { return f(x); };

app(sqr,4);     app(dbl,5);
function app(f,x) { return f(x); };

app(sqr,4);     app(dbl,5);
function app(f, x) { return f(x); }; 

app(sqr, 4); app(dbl1, 5);
function app(f, x) { return f(x); };

app(sqr, 4);  app(dbl, 5);

{sqr}
function app(f, x) { return f(x); };

app(sqr, 4);     app(dbl, 5);
function app(f, x) {
    return f(x);
};

app(sqr, 4);
app(dbl, 5);
function app(f, x) { return f(x); };

app(sqr, 4);     app(dbl, 5);

{app(sqr, 4)};    {app(dbl, 5)};

{app(sqr, 4)};    {app(dbl, 5)};

{app(sqr, 4)};    {app(dbl, 5)};
function app(f, x) { return f(x); }; 

app(sqr, 4);  
app(dbl, 5);  

{app sqr 4}  
{app dbl 5}  

{app sqr 4}  
{app sqr 4}  

{app sqr 4}
function app(f, x) {
    return f(x);
};

app(sqr, 4);
app(dbl, 5);

{sqrt}
{4}

app(sqr, 4);
app(dbl, 5);

{sqrt(4)}
function app(f, x) { return f(x); };

app(sqr, 4);     app(dbl, 5);

{sqr, dbl} {4}

app(sqr, 4);

app(dbl, 5);

{sqr(4)}
function app(f, x) { return f(x); };
function app(f, x) { return f(x); };

app(sqr, 4);     app(dbl, 5);

{sqr, dbl}   {4,5}

app(sqr, 4);    app(dbl, 5);

{sqr(4)}
function app(f, x) { return f(x); };

app(sqr, 4);     app(dbl, 5);

{sqr, dbl}  {4, 5}

app(sqr, 4);
app(dbl, 5);

{sqr(4), sqr(5),
    dbl(4), dbl(5)}
function app(f, x) { return f(x); };

app(sqr, 4);     app(dbl, 5);

{sq, db}  {4, 5}

app(sqr, 4);     app(dbl, 5);

{sq(4), sq(5),
  db(4), db(5)}
function app(f, x) { return f(x); }; 

app(sqr, 4);     app(dbl, 5); 

{4, 5} 

{sqr(4), sqr(5), 
  dbl(4), dbl(5)}

Theorem: 0CFA is complete for PTIME.
0CFA
Simple closure
Simple closure

```javascript
function app(f, x) { return f(x); };

app(sqr, 4);     app(dbl, 5);

{sqr, dbl}   {4,5}
```

```javascript
app(sqr, 4);
app(dbl, 5);

{sqr(4), sqr(5),
  dbl(4), dbl(5)}
```
Simple closure

```javascript
function app(f, x) { return f(x); };

app(sqr, 4);     app(dbl, 5);

{sqr, dbl} {4,5}

app(sqr, 4);     app(dbl, 5);

{sqr(4), sqr(5),
  dbl(4), dbl(5)}
```
function app(f, x) { return f(x); };

app(sqr, 4);     app(dbl, 5);

{4, 5}

{sqr, dbl}  {4, 5}

app(sqr, 4);     app(dbl, 5);

{sqr(4), sqr(5),
  dbl(4), dbl(5)}
function app(f, x) { return f(x); };

app(sqr, 4);     app(dbl, 5);

Theorem: Simple closure is complete for PTIME.

{app(sqr, 4), app(dbl, 5),
  {sqr(4), sqr(5),
   dbl(4), dbl(5)}}
0CFA
Simple closure
function app(f, x) { return f(x); }; 

app(sqr, 4);  
app(dbl, 5); 

{sqr}  {4}  

app(sqr, 4);  
app(dbl, 5); 

{sqr(4)}
function app(f, x) { return f(x); };

call 1: app(sqr, 4);

call 2: app(dbl1, 5);

call 3: app(sqr, 4)
function app(f, x) { return f(x); };
function app(f,x) { return f(x); };

app(sqr,4);

app(dbl,5);

{sqr(4)}
function app(f, x) { return f(x); };

app(sqr, 4);    app(dbl, 5);

{sqr(4)}
function app(f, x) { return f(x); };
function app(f, x) { return f(x); };
Sub0CFA

function app(f, x) { return f(x); };

app(sqr, 4);     app(dbl, 5);

Theorem: Sub0CFA is complete for PTIME.
function app(f,x) { return f(x); };

app(sqr,4);
app(dbl,5);
function app(f, x) { return f(x); };

app(sqr, 4);     app(dbl, 5);

Theorem: They’re all complete for PTIME.
Precision

0CFA
Simple closure
Sub0CFA
:
Precision

1CFA
0CFA
Simple closure
Sub0CFA
:
function app(f, x) { return f(x); };
function app(f, x) { return f(x); };

app(sqr, 4);     app(dbl, 5);

{sqr}    {4}

app(sqr, 4);
app(dbl, 5);

{sqr(4)}
function app(f, x) { return f(x); }; 

app(sqr, 4);     app(dbl, 5); 

{sqr}  {4} 

{dbl}  {5} 

app(sqr, 4);     app(dbl, 5); 

{sqr(4)}
function app(f, x) { return f(x); }; 

app(sqr, 4);     app(dbl, 5); 

{ sqr } { 4 } 

{ dbl } { 5 }
function app(f, x) { return f(x); };
function app(f, x) { return f(x); };

app(sqr, 4);     app(dbl, 5);

Theorem: 1CFA is complete for EXPTIME.
Precision

1CFA
0CFA
Simple closure
Sub0CFA

:
Precision

\[ k \text{CFA} \]

\[ \vdots \]

1CFA

0CFA

Simple closure

Sub0CFA

\[ \vdots \]
function app(f, x) { return f(x); };  

app(sqr, 4);  
app(dbl, 5);  

{kCFA}  

{app(sqr, 4);}  
{app(dbl, 5);}  
{sqr}  
{4}  
{dbl}  
{5}  
{sqr(4)}  
{dbl(5)}
Theorem: $k$CFA is complete for $\text{EXPTIME}$.
Precision

$kCFA$

$1CFA$

$0CFA$

Simple closure

$Sub0CFA$

$\vdots$
Precision

$\vdots$

$k$CFA

$1$CFA

$0$CFA

Simple closure

Sub$0$CFA

$\vdots$

EXPTIME

PTIME
Rigor (mortis) of existing analyses
the Semantic Gap

Table 3.1: Abstract Control Flow Analysis (Subsections 3.1.1 and 3.1.2).
the Semantic Gap
the Semantic Gap

\[ \text{Table 3.2: The Structural Operational Semantics of } \text{FUN (part 1).} \]
**the Semantic Gap**

### Table 3.1: Abstract Control Flow Analysis (Subsections 3.1.1 and 3.1.2).

<table>
<thead>
<tr>
<th>Var</th>
<th>( p \vdash x' \to v' ) if ( x \in \text{dom}(p) ) and ( v = p(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>[fn]</td>
<td>( p \vdash (\text{fn } x \Rightarrow e_0)' \to (\text{close } (\text{fn } x \Rightarrow e_0) \text{ in } p_0)' )</td>
</tr>
<tr>
<td>[fun]</td>
<td>( p \vdash (\text{fn } x \Rightarrow e_0)' \to (\text{close } (\text{fun } x \Rightarrow e_0) \text{ in } p_0)' )</td>
</tr>
<tr>
<td>[app]</td>
<td>( p \vdash (i_1 \ i_2)' \to (i_1' \ i_2') )</td>
</tr>
<tr>
<td>[app]</td>
<td>( p \vdash (i_1 \ i_2)' \to (i_1' \ i_2') )</td>
</tr>
<tr>
<td>[app]</td>
<td>( p \vdash ((\text{close } (\text{fun } x \Rightarrow e_1) \text{ in } p_0)' \to (\text{bind } p_0[x \to v_0] \text{ in } e_1)' )</td>
</tr>
<tr>
<td>[app]</td>
<td>( p \vdash ((\text{close } (\text{fun } x \Rightarrow e_1) \text{ in } p_0)' \to (\text{bind } p_0[x \to v_0] \text{ in } e_1)' )</td>
</tr>
<tr>
<td>[bind]</td>
<td>( p \vdash (i_1 \ i_2)' \to (i_1' \ i_2') )</td>
</tr>
<tr>
<td>[bind]</td>
<td>( p \vdash (\text{bind } p_1 \text{ in } i_1)' \to (\text{bind } p_1 \text{ in } i_1)' )</td>
</tr>
<tr>
<td>[bind]</td>
<td>( p \vdash (\text{bind } p_1 \text{ in } i_1)' \to (\text{bind } p_1 \text{ in } i_1)' )</td>
</tr>
</tbody>
</table>

### Table 3.2: The Structural Operational Semantics of FUN (part 1).
the Semantic Gap

\[ \rho \vdash ie_0 \rightarrow ie'_0 \]
\[ \rho \vdash (if \ ie_0 \ then \ e_1 \ else \ e_2)^\ell \rightarrow (if \ ie'_0 \ then \ e_1 \ else \ e_2)^\ell \]
\[ \rho \vdash (if \ true^{\ell_0} \ then \ t_1^{\ell_1} \ else \ t_2^{\ell_2})^\ell \rightarrow t_1^\ell \]
\[ \rho \vdash (if \ false^{\ell_0} \ then \ t_1^{\ell_1} \ else \ t_2^{\ell_2})^\ell \rightarrow t_2^\ell \]
\[ \rho \vdash ie_1 \rightarrow ie'_1 \]
\[ \rho \vdash (let \ x = ie_1 \ in \ e_2)^\ell \rightarrow (let \ x = ie'_1 \ in \ e_2)^\ell \]
\[ \rho \vdash (let \ x = v^{\ell_1} \ in \ e_2)^\ell \rightarrow (bind \ \rho_0[x \mapsto v] \ in \ e_2)^\ell \]
where \( \rho_0 = \rho \mid FV(e) \)
\[ \rho \vdash ie_1 \rightarrow ie'_1 \]
\[ \rho \vdash (ie_1 \ op \ ie_2)^\ell \rightarrow (ie'_1 \ op \ ie_2)^\ell \]
\[ \rho \vdash ie_2 \rightarrow ie'_2 \]
\[ \rho \vdash (v_1^{\ell_1} \ op \ ie_2)^\ell \rightarrow (v_1^{\ell_1} \ op \ ie_2)^\ell \]
\[ \rho \vdash (v_1^{\ell_1} \ op \ v_2^{\ell_2})^\ell \rightarrow v^\ell \quad \text{if } v = v_1 \ op \ v_2 \]

\begin{table}[h]
\centering
\caption{The Structural Operational Semantics of FUN (part 2).}
\begin{tabular}{ll}
\hline
\hline
[appr] & \hline
[appr] & \hline
[appr] & \hline
[appr] & \hline
[appr] & \hline
[bind] & \hline
[bind] & \hline
\hline
\end{tabular}
\end{table}
the Semantic Gap
the Semantic Gap
the Semantic Gap
the Semantic Gap
My challenge to ICFP:

Develop a program analysis for reasoning about:

- Space-consumption in a lazy language
- State and control in a language with effects
- Security in a language with stack inspection
- Blame in a language with behavioral contracts
- Safe parallelism in a language with futures
My challenge to ICFP:

Develop a program analysis for reasoning about:

- Space-consumption in a lazy language
- State and control in a language with effects
- Security in a language with stack inspection
- Blame in a language with behavioral contracts
- Safe parallelism in a language with futures
Modularity of existing analyses
Three approaches:
Three approaches:

Do nothing (analyze whole programs only)
Three approaches:

Do nothing (analyze whole programs only)

Hemorrhage precision (black hole approach)
Three approaches:

Do nothing (analyze whole programs only)

Hemorrhage precision (black hole approach)

Do something really complicated
Modular Set-Based Analysis from Contracts

Philippe Meunier
College of Computer and Information Science, Northeastern University
meunier@ccs.neu.edu

Robert Bruce Findler
Department of Computer Science, University of Chicago
robby@cs.uchicago.edu

Matthias Felleisen
College of Computer and Information Science, Northeastern University
matthias@ccs.neu.edu

Abstract
In PLT Scheme, programs consist of modules with contracts. The latter describe the inputs and outputs of functions and objects via predicates. A run-time system enforces these predicates; if a predicate fails, the enforcer raises an exception that blames a specific module with an explanation of the fault.

In this paper, we show how to use such module contracts to turn set-based analysis into a fully modular parameterized analysis. Using this analysis, a static debugger can indicate for any given contract check whether the corresponding predicate is always satisfied, partially satisfied, or (potentially) completely violated. The static debugger can also predict the source of potential errors, i.e., it is sound with respect to the blame assignment of the contract system.

Categories and Subject Descriptors F.3.2 [Semantics of Programming Languages]: Program analysis; D.2.4 [Software/Program Verification]: Programming by contract

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Keywords: Static Debugging, Set-based Analysis, Modular Analysis, Runtime Contracts.

1. Modules, Contracts, and Static Debugging
A static debugger helps programmers find errors via program analyses. It uses the invariants of the programming language to analyze the program and determines whether the program may violate one of them during execution. For example, a static debugger can find expressions that may dereference null pointers. Some static debuggers use lightweight analyses, e.g., Flanagan et al.'s MrSpidey [11] relies on a variant of set-based analysis [10, 16, 21]; others use a deep abstract interpretation, e.g., Bourdoncle's Syntox [4]; and yet others employ theorem proving, e.g., Deliot et al.'s ESC [7].

Experience with static debuggers shows that they work well for reasonably small programs. Using MrSpidey, we have routinely debugged or re-engineered programs of 2,000 to 5,000 lines of code in PLT Scheme. Flanagan has successfully analyzed the core of the interpreter, dubbed MrEd [13], a 40,000 line program. Existing static debuggers, however, suffer from a monolithic approach to program analysis. Because their analyses require the availability of the entire program, programmers cannot analyze their programs until they have everyone else’s modules.

Permission to copy beyond the scope of this license requires the express permission of the author(s) and of ACM.

Over the past few years, we have added a first-order module system to PLT Scheme [12] and have equipped the module system with a contract system [8]. A contract is roughly a predicate on the inputs and outputs of (exported) functions, including object methods and higher-order functions. The contract system monitors the contracts during program execution. If a module violates a contract, the contract system pinspoints the guilty party and issues an explanatory message.

This paper makes five contributions to static debugging and software contracts. First, it explains how to construct a modular static debugger for programs with contracts, using those contracts in a dual role: one as a source of abstract values and one as a sink for abstract values. Second, we prove that our contract-based, whole-program analysis computes its results in a modular manner. That is, our contract-aware set-based analysis produces the same predictions for a given point in the program regardless of whether it analyzes the whole program or just the surrounding module. Third, for any given contract check, the system indicates whether the corresponding predicate is always satisfied, partially satisfied, or completely violated. Fourth, the static debugger can also predict the source of potential errors, i.e., it is sound with respect to the blame assignment of the contract system. Fifth, the analysis is parameterized over both a predicate approximation relation and a predicate domain function.

2. Overview
The paper presents a model of a modular static debugger. The model consists of two parts: a runtime contract system and a set-based analysis for modules with contracts. A correctness theorem ties the two parts together. Figure 1 provides an overview of these three pieces in graphical form. The vertical column on the left represents the runtime contract system. A contract compiler translates a collection of modules and a main expression into a suitably annotated form. During execution, which we naturally model via a reduction system, the contract system keeps track of the contract obligations; if something goes wrong it blames a specific module.

The first horizontal row of Figure 1 depicts the analysis process, which consists of three stages. First, it partitions the program into module-like pieces by lifting expressions with contract annotations out of the main program. Second, the resulting collection of program pieces is analyzed with a parameterized set-based analysis. This step yields both sets of abstract values and sets of potential errors, including explanations that blame the guilty party: we call the latter blame sets. Third, the former are summarized as set-of-values descriptions, dubbed types.

The rest of the grid in Figure 1 explains our proof technique for the correctness theorem. Since each reduction step creates a complete program, the correctness proof can proceed via subject reduction. We reapply the analysis after each reduction step. The proof then shows that the reductions preserve the types and the blame...
Table 1. Constraints creation for source-sink pairs.

<table>
<thead>
<tr>
<th>Source\Sink</th>
<th>( (a,c) )</th>
<th>( (a,e) )</th>
<th>( (a,f) )</th>
<th>( (b,c) )</th>
<th>( (b,e) )</th>
<th>( (b,f) )</th>
<th>( (c,e) )</th>
<th>( (d,e) )</th>
<th>( (d,f) )</th>
<th>( (e,f) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
</tr>
<tr>
<td>2. ( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
</tr>
<tr>
<td>3. ( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
<td>( (\mathcal{P}_1 \setminus h)^+ ) ( \subseteq (\mathcal{P} \setminus h)^+ )</td>
</tr>
</tbody>
</table>
Semantics-based analysis matters.

A static debugger helps programmers find errors via program analysis. To incorporate program analysis into a debugger, a contract system is used. A contract is roughly a predicate on the contracts during program execution. If a module violates a contract and its labels are added, a predicate domain function is used to simplify the soundness proof.

Table 1. Constraints creation for source-sink pairs.

<table>
<thead>
<tr>
<th>Source \ Sink</th>
<th>Constraint</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(\alpha \in {e, f} \land \phi \in {\lambda, \omega}) \land \psi = 0</td>
<td>(\alpha \in {e, f} \land \phi \in {\lambda, \omega}) \land \psi = 0</td>
</tr>
<tr>
<td>2</td>
<td>(\alpha \in {e, f} \land \phi \in {\lambda, \omega}) \land \psi = 0</td>
<td>(\alpha \in {e, f} \land \phi \in {\lambda, \omega}) \land \psi = 0</td>
</tr>
<tr>
<td>3</td>
<td>(\alpha \in {e, f} \land \phi \in {\lambda, \omega}) \land \psi = 0</td>
<td>(\alpha \in {e, f} \land \phi \in {\lambda, \omega}) \land \psi = 0</td>
</tr>
<tr>
<td>4</td>
<td>(\alpha \in {e, f} \land \phi \in {\lambda, \omega}) \land \psi = 0</td>
<td>(\alpha \in {e, f} \land \phi \in {\lambda, \omega}) \land \psi = 0</td>
</tr>
<tr>
<td>5</td>
<td>(\alpha \in {e, f} \land \phi \in {\lambda, \omega}) \land \psi = 0</td>
<td>(\alpha \in {e, f} \land \phi \in {\lambda, \omega}) \land \psi = 0</td>
</tr>
<tr>
<td>6</td>
<td>(\alpha \in {e, f} \land \phi \in {\lambda, \omega}) \land \psi = 0</td>
<td>(\alpha \in {e, f} \land \phi \in {\lambda, \omega}) \land \psi = 0</td>
</tr>
<tr>
<td>7</td>
<td>(\alpha \in {e, f} \land \phi \in {\lambda, \omega}) \land \psi = 0</td>
<td>(\alpha \in {e, f} \land \phi \in {\lambda, \omega}) \land \psi = 0</td>
</tr>
<tr>
<td>8</td>
<td>(\alpha \in {e, f} \land \phi \in {\lambda, \omega}) \land \psi = 0</td>
<td>(\alpha \in {e, f} \land \phi \in {\lambda, \omega}) \land \psi = 0</td>
</tr>
</tbody>
</table>

Figure 1. Overview of these modules and their interactions.
Higher-order Program Analysis is Alive and Well.

(I have a way forward.)
A Way Forward

Scalability

Complexity
Maintenance
Verification
Expressivity
Modularity
Key insight: analysis is a kind of evaluation
Every variable occurs once.
Every variable occurs once.
Every variable occurs once.
Every variable occurs once.
Datalog-style programming with analysis.

$k\text{CFA}$  

Eval
$kCFA \quad EXPTIME \quad \text{Eval}$
Similar precision, better performance
Precision

\[ k_{\text{CFA}} \]
\[ \vdots \]
\[ 1_{\text{CFA}} \]
\[ 0_{\text{CFA}} \]
\[ \text{Sub} 0_{\text{CFA}} \]
\[ \text{Simple closure} \]
\[ \vdots \]

EXPTIME

PTIME
Precision

0CFA
Sub0CFA
Simple closure

PTIME

EXPTIME

PTIME

\(kCFA\) \(mCFA\)

\(1CFA\) \(1CFA\)

\(\vdots\) \(\vdots\)
A Systematic Approach to Program Analysis Design
Semantics → Analysis
Semantics → Analysis
Semantics

Machine

Analysis
Analysis machine

\[ e \xrightarrow{} S_0 \xrightarrow{} S_1 \xrightarrow{} S_2 \xrightarrow{} S_3 \xrightarrow{} S_4 \xrightarrow{} \ldots \]
Analysis machine

e
\[ S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow \ldots \]
Analysis machine

$e$

$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow \ldots$

$\hat{S}_0$
Analysis machine

$e \rightarrow S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow \ldots$

$\hat{S}_0 \rightarrow \hat{S}_1 \rightarrow \hat{S}_2 \rightarrow \hat{S}_3 \rightarrow \hat{S}_{3.1}$
Analysis machine

e
$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow \ldots$

$\hat{S}_0 \rightarrow \hat{S}_1 \rightarrow \hat{S}_2 \rightarrow \hat{S}_3 \rightarrow \hat{S}_4$

$\hat{S}_3.1$
Analysis machine

\[ e \]

\[ S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow \ldots \]

\[ \hat{S}_0 \rightarrow \hat{S}_1 \rightarrow \hat{S}_2 \rightarrow \hat{S}_3 \rightarrow \hat{S}_4 \]

\[ \hat{S}_{3.1} \]
Theorem: The analysis simulates the machine.
Key idea:

Deterministic state transition system with an infinite state space.

Non-deterministic state transition system with a finite state space.
Key idea:

Deterministic state transition system with an infinite state space.

Non-deterministic state transition system with a finite state space.
Program analysis...

Infinite state-space
Program analysis...

Infinite state-space

Finite state-space
...is bounded graph search.
...is bounded graph search.

Finite state-space
Semantics

A Closer Look

Machine

Analysis
Reduction semantics:

Syntax: \[ e ::= n \mid x \mid (e+e) \]

Reduction: \[(n+m) \rightarrow n + m\]
\[x \rightarrow n \text{ where } \rho(x) = n\]

Eval. Contexts: \[ E ::= \[\] \mid (E+e) \mid (n+E) \]
Reduction semantics:
\[ \ldots \]
\[ E[(x+y)] \]
$E[[]]$
\begin{align*}
E[([] + y)] & \\
x & \rightarrow 3
\end{align*}
\( E[([ ]+y)] \)

\[ x \rightarrow 3 \]
\[ E\left(\left[3\right]+y\right) \]
\( \langle e, C \rangle \) \( \rightarrow \) \( \langle 7, E[\] \( \rangle \)

\[ \sigma(a) = C \]

\[ \langle n, C \rangle \rightarrow \langle e_1, \rho, \sigma, C \rangle \]

\[ \langle e_1, \rho, \sigma, c_1(C, e_2) \rangle \]

\[ \langle e_1, \rho, \sigma, a \rangle \rightarrow \langle e_1, \rho, \sigma, a' \rangle \]

\[ \langle e_1, \rho, \hat{\sigma}, a \rangle \rightarrow \langle e_1, \rho, \hat{\sigma} \cup \{a' \mapsto \to c_1(a, e_2)\}, a' \rangle \]

\[ \langle n, C \rangle \rightarrow \langle e_1, \rho, \sigma, C \rangle \]

where \( \rho(x) = n \)

\[ \langle n, c_1(C, e_2) \rangle \rightarrow \langle e, c_2(n, C) \rangle \]

\[ \langle m, c_2(n, C) \rangle \rightarrow \langle n + m, C \rangle \]
\[ E[(3+y)] \]
\[ E[(3 + [ ]) ] \]

\[ y \rightarrow 4 \]
\[
E[(3 + [0])] = 4
\]
$E[(3+[4])]$
\[ \langle e, C \rangle \]
\( E[ (3+4) ] \)
\[ E[\ ] \quad (3+4) \rightarrow 7 \]
Reduction semantics:

Syntax: \[ e ::= n \mid x \mid (e+e) \]

Reduction: \[(n+m) \rightarrow n + m\]
\[x \rightarrow n \text{ where } \rho(x) = n\]

Eval. Contexts: \[ E ::= [ ] \mid (E+e) \mid (n+E) \]
Reduction semantics:

Syntax:  \[ e ::= n \mid x \mid (e+e) \]

Reduction:  \[(n+m) \rightarrow n + m \]

\[ x \rightarrow n \text{ where } \rho(x) = n \]

Eval. Contexts:  \[ E ::= [ ] \mid (E+e) \mid (n+E) \]

Continuations:  \[ C ::= c_0 \mid c_1(e, C) \mid c_2(n, C) \]
\[ E ::= \emptyset \mid (E + e) \mid (n + E) \]

\[ C ::= c_0 \mid c_1(C, e) \mid c_2(n, C) \]

\[ (x + y) \]
\[ c_1(C, y) \]
\[ c_2(C, 3) \]

\[ y \rightarrow 4 \]
(3+4) → 7
Stack machine:

\[
\begin{align*}
\langle (e_1 + e_2), C \rangle & \rightarrow \langle e_1, c_1(C, e_2) \rangle \\
\langle x, C \rangle & \rightarrow \langle n, C \rangle \quad \text{where } \rho(x) = n \\
\langle n, c_1(C, e) \rangle & \rightarrow \langle e, c_2(n, C) \rangle \\
\langle m, c_2(n, C) \rangle & \rightarrow \langle n + m, C \rangle
\end{align*}
\]

Correctness:

\[
P \rightarrow^* n \iff \langle P, c_0 \rangle \rightarrow^* \langle n, c_0 \rangle
\]
Syntax:

\[
c ::= \text{num} \mid \text{str} \mid \text{bool} \mid \text{undefined} \mid \text{null}
\]

\[
v ::= c \mid \text{func}(\vec{x}) \{ \text{return } e \} \mid \{ \text{str}.v \ldots \}
\]

\[
p ::= \text{str} : e
\]

\[
e ::= x \mid v \mid \{ \vec{p} \} \mid \text{let } (x = e) e \mid e(\vec{e})
\]

\[
| e[e] \mid e[e] = e \mid \text{del } e[e]
\]
Reductions:

\[
\text{let } (x = v) e \rightarrow [v/x]e
\]

\[
\text{func}(\overline{x}) \{ \text{return } e \}(\overline{v}) \rightarrow [\overline{v}/\overline{x}]e
\]

\[
\{ \ldots str_i.v \ldots \}[str_i] \rightarrow v
\]

\[
str_x \notin (str_1 \ldots)
\]

\[
\{ str_1.v_1 \ldots \}[str_x] \rightarrow \text{undefined}
\]

\[
\{ \ldots str_i.v_i \ldots \}[str_i] = v \rightarrow \{ \ldots str_i.v \ldots \}
\]

\[
str_x \notin (str_1 \ldots)
\]

\[
\{ str_1.v_1 \ldots \}[str_x] = v \rightarrow \{ str_x.v, str_1.v_1 \ldots \}
\]

\[
\text{del } \{ \ldots str_i.v_i \ldots \}[str_i] \rightarrow \{ \ldots \}
\]

\[
str_x \notin (str_1 \ldots)
\]

\[
\text{del } \{ str_1.v_1 \ldots \}[str_x] \rightarrow \{ str_1.v_1 \ldots \}
\]
Eval. Contexts:

\[
E ::= [ ] \\
| \text{let } (x = E') \ e \\
| E(e) \\
| v(e \ldots E, v \ldots) \\
| \{ \text{str} : v \ldots \text{str} : E, \vec{p} \} \\
| E[e] \\
| v[E] \\
| E[e] = e \\
| v[E] = e \\
| v[v] = E \\
| \text{del } E[e] \\
| \text{del } v[E]
\]

Continuations:

\[
C ::= c_1 \\
| c_2(x, e, \rho, C) \\
| c_3(\vec{e}, \rho, C) \\
| c_4(c, \vec{c}, \vec{e}, \rho, C) \\
| c_5(\text{str}, \vec{q}, \vec{p}, \rho, C) \\
| c_6(e, \rho, C) \\
| c_7(c, C) \\
| c_8(e, e, \rho, C') \\
| c_9(c, e, \rho, C') \\
| c_{10}(c, c, C) \\
| c_{11}(e, \rho, C) \\
| c_{12}(c, C)
\]
Machine:

\[
\begin{align*}
\langle x, \rho \rangle, \sigma, C' & \quad \rightarrow \quad \langle c, \sigma, C \rangle \\
\langle c, \sigma, c_2(x, e, \rho, C) \rangle & \quad \rightarrow \quad \langle e, \rho[x \mapsto a], \sigma[a \mapsto c], C' \rangle \\
\langle c, \sigma, c_4(\langle \text{func}(x) \{ \text{return } e \} , \rho \rangle, c_n \ldots c_0 \ldots, \rho', C) \rangle & \quad \rightarrow \quad \langle e, \rho[\vec{x} \mapsto \vec{a}], \sigma[\vec{a} \mapsto c_0 \ldots c_n c], C' \rangle \\
\langle \text{str}_x, \rho \rangle, \sigma, c_7(\{ \text{str}_1, c_1 \ldots \}, C) & \quad \rightarrow \quad \langle c_1, \sigma, C \rangle \\
\langle c, \sigma, c_{10}(\{ \ldots \text{str}_i, c_i \ldots \}, \langle \text{str}_i, \rho \rangle, C) \rangle & \quad \rightarrow \quad \langle \text{undefined}, \sigma, C \rangle \\
\langle \text{str}_x, \rho \rangle, \sigma, c_{12}(\{ \ldots \text{str}_i, c_i \ldots \}, C) & \quad \rightarrow \quad \langle \text{str}_x \cdot c, \text{str}_1 \cdot c_1 \ldots \}, \sigma, C \rangle \\
\langle c, \sigma, c_3(e\vec{c}, \rho, C) \rangle & \quad \rightarrow \quad \langle \text{str}_x, \rho \rangle, \sigma, C_{12}(\{ \text{str}_1 \cdot c_1 \ldots \}, C) \\
\langle \text{func}(\) \{ \text{return } e \} , \rho \rangle, \sigma, c_3(\), \rho', C) & \quad \rightarrow \quad \langle e, \rho, \sigma, c_4(c, e\vec{c}, \rho, C) \rangle \\
\langle c, \sigma, c_5(\text{str}, \bar{q}, \rho, C) \rangle & \quad \rightarrow \quad \langle \text{let} (x = e_0) \ e_1, \rho \rangle, \sigma, C' \\
\langle e() \rangle, \rho, \sigma, C \rangle & \quad \rightarrow \quad \langle e, \rho, \sigma, C \rangle \\
\langle e_0(e\vec{c}), \rho, \sigma, C \rangle & \quad \rightarrow \quad \langle e_0(e\vec{c}), \rho, \sigma, C \rangle \\
\langle \{ \), \rho \rangle, \sigma, C \rangle & \quad \rightarrow \quad \langle \{ \), \rho \rangle, \sigma, C \rangle \\
\langle \{ \text{str}_0 : e_0 \bar{p} \}, \rho \rangle, \sigma, C \rangle & \quad \rightarrow \quad \langle \{ \text{str}_0 : e_0 \bar{p} \}, \rho \rangle, \sigma, C \rangle \\
\langle e_0[e_1], \rho \rangle, \sigma, C \rangle & \quad \rightarrow \quad \langle e_0[e_1], \rho \rangle, \sigma, C \rangle \\
\langle e_0[e_1] = e_2, \rho \rangle, \sigma, C \rangle & \quad \rightarrow \quad \langle e_0[e_1] = e_2, \rho \rangle, \sigma, C \rangle \\
\langle \text{del} e_0[e_1], \rho \rangle, \sigma, C \rangle & \quad \rightarrow \quad \langle \text{del} e_0[e_1], \rho \rangle, \sigma, C \rangle \\
\end{align*}
\]

where $\sigma(\rho(x)) = c$

where $a$ is fresh

where $\vec{a}$ are fresh

where $\text{str}_x \notin (\text{str}_1 \ldots )$

where $\text{str}_x \notin (\text{str}_1 \ldots )$

where $\text{str}_x \notin (\text{str}_1 \ldots )$
\[ \langle e, C \rangle \]
\langle e, \rho, \sigma, C \rangle

\text{Var} \rightarrow \text{Addr}

\text{Addr} \rightarrow \text{Value}
Semantics

Machine

A Closer Look

Analysis
Semantics

A Closer Look

Machine

Analysis
Key idea:

Deterministic state transition system with an infinite state space.

Non-Deterministic state transition system with a finite state space.
Step 1:

\[
\langle e, \rho, \sigma, C \rangle
\]

Move continuations into heap.
Step 1:

\[ \langle e, \rho, \sigma, C \rangle \]

Move continuations into heap.
Step 1:

\[ \langle e, \rho, \sigma, a \rangle \]

\[ \sigma(a) = C \]
Step 1:  
\[
\langle e, \rho, \sigma, a \rangle
\]
\[
\sigma(a) = C
\]
Step 1:

\[\langle (e_1 + e_2), \rho, \sigma, C \rangle \rightarrow \langle e_1, \rho, \sigma, c_1(C, e_2) \rangle \]

\[\iff\]

\[\langle (e_1 + e_2), \rho, \sigma, a \rangle \rightarrow \langle e_1, \rho, \sigma[a' \mapsto c_1(a, e_2)], a' \rangle \]
Step 2:

\[\langle e, \rho, \hat{\sigma}, a \rangle\]

\[\hat{\sigma}(a) \ni C\]
Step 2:

\[ \langle e, \rho, \hat{\sigma}, a \rangle \]

\[ \hat{\sigma}(a) \ni C \]
Step 1:

\[
\langle (e_1+e_2), \rho, \sigma, C \rangle \rightarrow \langle e_1, \rho, \sigma, c_1(C, e_2) \rangle
\]

\[
\iff
\]

\[
\langle (e_1+e_2), \rho, \sigma, a \rangle \rightarrow \langle e_1, \rho, \sigma[a' \mapsto c_1(a, e_2)], a' \rangle
\]

\[
\implies
\]

\[
\langle (e_1+e_2), \rho, \hat{\sigma}, a \rangle \rightarrow \langle e_1, \rho, \hat{\sigma} \sqcup [a' \mapsto c_1(a, e_2)], a' \rangle
\]
Semantics Engineering with PLT Redex

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt
Analysis of:
First-class control
Exceptions
Mutation
Base values

Machine

Analysis
Semantics


James Gosling • Bill Joy • Guy Steele • Gilad Bracha

The Java™ Series

...from the Source

CD-ROM Included

Sun Microsystems
Static verification of security via stack inspection
Abstract Models of Memory Management*

Greg Morrisett  
Carnegie Mellon  
jgmorris@cs.cmu.edu

Matthias Felleisen  
Rice University  
matthias@cs.rice.edu

Robert Harper  
Carnegie Mellon  
rwb@cs.cmu.edu

Abstract

Most specifications of garbage collectors concentrate on the low-level algorithmic details of how to find and preserve accessible objects. Often, they focus on bit-level manipulations such as "scanning stack frames," "marking objects," "tagging data," etc. While these details are important in some contexts, they often obscure the more fundamental aspects of memory management: what objects are garbage and why?

We develop a series of calculi that are just low-level enough that we can express allocation and garbage collection, yet are sufficiently abstract that we may formally prove the correctness of various memory management strategies. By making the heap of a program syntactically apparent, we can specify memory actions as rewriting rules that allocate values on the heap and automatically dereference pointers to such objects when needed. This formulation permits the specification of garbage collection as a relation that removes portions of the heap without affecting the outcome of the evaluation.

Our high-level approach allows us to specify in a compact manner a wide variety of memory management techniques, including standard trace-based garbage collection (i.e., the family of copying and mark/sweep collection algorithms), generational collection, and type-based, tag-free collection. Furthermore, since the definition of garbage is based on the semantics of the underlying language instead of the conservative approximation of inaccessibility, we are able to specify and prove the idea that type inference can be used to collect some objects that are accessible but never used.

* This work was sponsored in part by the Advanced Research Projects Agency (ARPA), CSTO, under the title "The Fox Project: Advanced Development of Systems Software." ARPA Order No. 8313, issued by ESD/AVR under Contract No. F04622–89–C–0168, Wright Laboratory, Aeronautical Systems Center, Air Force Material Command, USAF, and ARPA grant No. F33615–91–C–1130. Views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of Wright Laboratory or the United States Government.

Permission to make digital/hard copies of all or part of this material is granted provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright notice, the title of the publication and its date appear, and notice is given that copyright is reserved by ACM. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires specific permission and/or fee.

ICFP '10/CACM '11

1 Memory Safety

Advanced programming languages manage memory allocation and deallocation automatically. Automatic memory managers, or garbage collectors, significantly facilitate the programming process because programmers can rely on the language implementation for the delicate tasks of finding and freeing unneeded objects. Indeed, the presence of a garbage collector ensures memory safety in the same way that a type system guarantees type safety: no program written in an advanced programming language will crash due to dangling pointer problems while allocation, access, and deallocation are transparent. However, in contrast to type systems, memory management strategies and particularly garbage collectors rarely come with a compact formulation and a formal proof of soundness. Since garbage collectors work on the machine representations of abstract values, the very idea of providing a proof of memory safety sounds unrealistic given the lack of simple models of memory operations.

The recently developed syntactic approaches to the specification of language semantics by Pelleisen and Blel [17] and Mason and Talcott [18, 19] are the first execution models that are intentionally enough to permit the specification of memory management actions and yet are sufficiently abstract to permit compact proofs of important properties. Starting from the λ–μ calculus of Pelleisen and Blel, we design compact specifications of a number of memory management ideas and prove several correctness theorems.

The basic idea underlying the development of our garbage collection calculi is the representation of a program's run-time memory as a global series of syntactic declarations. The program evaluation rules allocate large objects in the global declaration, which represents the heap, and automatically dereference pointers to such objects when needed. As a result, garbage collection can be specified as any relation that removes portions of the current heap without affecting the result of a program's execution.

In Section 2, we present a small functional programming language, Agc, with a rewriting semantics that makes allocation explicit. We define a semantic notion of garbage collection for Agc and prove that there is no optimal collection strategy that is computable. In Section 3, we specify the "free-variable" garbage collection rule which models trace-based collectors including mark/sweep and copying collectors. We prove that the free-variable rule is correct and provide two "implementations" at the syntactic level: the first corresponds to a copying collector, the second to a generational one.

In Section 4, we formalize so-called "tag-free" collection algorithms for explicitly-typed, monomorphic languages such as Pascal and Algol [7, 29, 39]. We show how to recover
Abstract Models of Memory Management

Abstract: We consider a variety of approaches to construe a program's heap in a context-free and unambiguous manner. Our approach is based upon the notion of a "trace" for each object's lifecycle, defined as the sequence of events occurring in the program's execution that affect the object's state.

We demonstrate that using this approach, one can develop an automated tool for manipulating the program's heap. This tool allows programmers to view and interact with the program's memory in a way that is both intuitive and efficient.

Conclusions: Our experiments show that our approach is both practical and effective in managing memory. We believe that our work opens up new possibilities for the automation of memory management in programming languages.

Analysis

The analysis section of the paper presents the results of our experiments, as well as a discussion of the implications of our findings for the field of memory management.

Acknowledgments

We would like to thank the members of the research group for their contributions to this work. This research was supported by the National Science Foundation under grant number 12345678.

References


Improved precision and efficiency via abstract GC
Effects of abstract GC
Effects of abstract GC
Effects of abstract GC
My challenge to ICFP:

Develop a program analysis for reasoning about:

✓ Space-consumption in a lazy language
✓ State and control in a language with effects
✓ Security in a language with stack inspection
✓ Blame in a language with behavioral contracts
✍ Safe parallelism in a language with futures
✓ Space-consumption in a lazy language
✓ State and control in a language with effects
✓ Security in a language with stack inspection
✓ Blame in a language with behavioral contracts
✍ Safe parallelism in a language with futures
✓ Garbage collection
✓ Java
✓ JavaScript
✍ May happen in parallel for threads
Complexity and Modularity
Aho et al. (1968) Introduces the class “2NPDA”

Gives a $O(n^3)$ algorithm for 2NPDA.
Introduces the class "2NPDA"

Gives a $O(n^3)$ algorithm for 2NPDA.

IM LOC

+ $O(n^3)$ = 😞
On the Cubic Bottleneck in Subtyping and Flow Analysis

Nevin Heintze*  David McAllester†

Abstract

We prove that certain data-flow and control-flow problems are 2NPDA-complete. This means that these problems are in the class 2NPDA and that they are hard for that class. The fact that they are in 2NPDA demonstrates the richness of the class. The fact that they are hard for 2NPDA can be interpreted as evidence they can not be solved in sub-cubic time — the cubic time decision procedure for an arbitrary 2NPDA problem has not been improved since its discovery in 1968.

1. Introduction

Cubic time complexity has become a common feature of algorithms for the automated analysis of computer programs. There is a general feeling that many of these algorithms are inherently cubic time — no sub-cubic procedure has been found. Such cubic time algorithms include Shivers' control flow analysis [17], the Palsberg and O'Keefe method of determining typability in the Amadio-Cardelli type system [15, 1], and various set-based analyses [5, 10, 11]. At an intuitive level the inherent cubic complexity in all these problems arises from the need to compute a dynamic transitive closure — one must compute the transitive closure of a directed graph while adding edges to the input graph as a consequence of edges derived for the output graph. Not only do these problems all seem inherently cubic, they all seem structurally similar and inherently cubic for the same reason.

In order to better understand the "cubic bottleneck" in flow analysis, Melski and Reps have investigated a simple data-flow reachability problem [13]. They relate this data-flow reachability problem to the problem of contextfree-language reachability (CFL-reachability). An instance of the CFL-reachability problem consists of a context free grammar and a directed graph where each arc is labeled with a symbol from the terminal alphabet. The problem is to determine whether there is a path between two given nodes such that the sequence of labels on the arcs in that path is a string in the language generated by the given grammar. The CFL-reachability problem can be solved in \(O(|G|n^3)\) time where \(|G|\) is the size of the grammar (the number of productions in a Chomsky normal form grammar) and \(n\) is the number of nodes in the graph. Melski and Reps give a linear time reduction from data-flow reachability to CFL-reachability. This reduction produces a grammar of size \(n\), so the reduction appears to yield an \(O(n^3)\) method of solving data-flow reachability. However, Melski and Reps show that the reduction produces problems with special structure and that the overall running time of solving a data-flow problem by reduction to CFL-reachability is \(O(n^6)\). More significantly, Melski and Reps give a reduction of CFL-reachability to data-flow reachability which runs in \(O(|G|n)\) time. For a fixed grammar this reduction is linear time. If the data-flow reachability problem could be solved in sub-cubic time then the CFL-reachability problem over a fixed grammar could also be solved in sub-cubic time.

Here we investigate the cubic bottleneck by relating it to the class 2NPDA. 2NPDA is the class of languages (or problems) definable by a two way nondeterministic push-down automata. In 1968 it was shown that any problem in the class 2NPDA can be solved in cubic time [2]. But no sub-cubic procedure for an arbitrary 2NPDA problem is known. Neal has shown that a certain 2NPDA problem — ground monadic rewriting reachability (GMR-reachability) — is 2NPDA complete [14].2 In other words, this problem is both in the class 2NPDA and is 2NPDA-hard, i.e., if GMR-reachability can be solved in sub-cubic time then all 2NPDA problems can be solved in sub-cubic time. We review Neal's result here. We also show that data-flow reachability, control-flow reachability, and the complement of Amadio-

---

*Bell Labs, 600 Mountain Ave, Murray Hill, NJ 07974, nch@bell-labs.com.
†AT&T Labs, 600 Mountain Ave, Murray Hill, NJ 07974, dmac@research.att.com.

1Following Heintze and Jaffar [4], Melski and Reps formulate this data-flow reachability problem as a set-constraint problem. We use the data-flow formulation here because it seems closer to applications.

2Neal uses a "monotone closure" formulation of GMR-problem. We find the GMR formulation more natural.
Subcubic algorithms for recursive state machines:

Uses Rytter’s technique to obtain $O\left(\frac{n^3}{\log n}\right)$ CFL-reachability algorithm.

$\text{IM LOC}$

$+ O\left(\frac{n^3}{\log n}\right) = \text{sad face}$
IM LOC + $O(n^{2.9}) = ☹
1M LOC + \( O(n^{2.9}) \) = ☹
IK LOC
...of whole programs

Semantics → Analysis
...of whole programs

Semantics

Analysis

...of whole programs
Semantics

Analysis
...of **partial** programs
...of partial programs

Semantics

Analysis

...of partial programs

arXiv 1103.1362
$f(5)$
$f(5)$

$f: \text{int} \rightarrow \text{int}$
\( f(5) \rightarrow 120 \)
$f(5)$

$f : \text{int} \rightarrow \text{int}$
$f(5)$

$f: \text{int} \rightarrow \text{int}$
\[ f \rightarrow (\text{int} \rightarrow \text{int}) \]
\[ f \rightarrow (\text{int} \rightarrow \text{int}) \]
\[ (\text{int} \rightarrow \text{int})(5) \rightarrow \text{int} \]
Syntax:

\[
E ::= [ ] \mid (E + e) \mid (v + E)
\]

\[
e ::= v \mid x \mid (e + e)
\]

\[
v ::= n \mid \text{int}
\]

Reduction:

\[
(m + n) \rightarrow m + n
\]

\[
(\text{int} + v) \rightarrow \text{int}
\]

\[
(v + \text{int}) \rightarrow \text{int}
\]

Eval. Contexts:
Syntax: \[ e ::= \ldots \mid (\text{if}0 \ e \ e \ e) \]

Reduction: \[(\text{if}0 \ 0 \ e_1 \ e_2) \rightarrow e_1\]
\[(\text{if}0 \ n \ e_1 \ e_2) \rightarrow e_2 \text{ where } n \neq 0\]

Eval. Contexts:
\[ E ::= [ ] \mid (E+e) \mid (v+E) \mid (\text{if}0 \ E \ e \ e) \]
Syntax: \[ e ::= \ldots \mid (\text{if0 } e \ e \ e) \]

Reduction: \begin{align*}
(\text{if0 } 0 \ e_1 \ e_2) & \rightarrow e_1 \\
(\text{if0 } n \ e_1 \ e_2) & \rightarrow e_2 \text{ where } n \neq 0 \\
(\text{if0 } \text{int} \ e_1 \ e_2) & \rightarrow e_1 \\
(\text{if0 } \text{int} \ e_1 \ e_2) & \rightarrow e_2
\end{align*}

Eval. Contexts: \[ E ::= \Box \mid (E+e) \mid (v+E) \mid (\text{if0 } E \ e \ e) \]
Key idea:

Non-deterministic state transition system with an infinite state space.

Non-deterministic state transition system with a finite state space.
Scales to higher-order behavioral contracts

\[ f(5) \]

\[ f: \text{prime}? \rightarrow \text{int} \]
Scales to higher-order behavioral contracts

\[ f(5) \]

\[ f: \text{prime}\? \rightarrow \text{int} \]
Scales to higher-order behavioral contracts
Scales to higher-order behavioral contracts

\[ f \rightarrow (\text{prime} \Rightarrow \text{int}) \]
Scales to higher-order behavioral contracts

\[ f \rightarrow (\text{prime?} \rightarrow \text{int}) \]
\[ (\text{prime?} \rightarrow \text{int})(5) \rightarrow * \text{ int} \]
A Way Forward

Scalability

- Complexity
- Maintenance
- Verification
- Expressivity
- Modularity
Past

Complexity:
- ICFP’07: PTIME of context-insensitive CFA
- SAS’08: PTIME of sub-OCFAs
- ICFP’08: EXPTime of context-sensitive
- HOSC’11: Subcubic bottleneck broken

A Way Forward

Expressive, maintainable, verifiable, modular, performant:
- ICFP’10, CACM’11: Systematic approach analysis
- PLDI’10: Object-oriented, functional bridge
- SFP’10: Pushdown machine analysis
- 2011 (in prep): Modular reduction for modular analysis
Future

Scalability

Compositional
Componential
Modular
Parallel
Applied
Compositional
Composing analyses for mutual benefit
Componential analyses for separate analysis

Modular
Beyond types and contracts as specifications

Parallel
May happen in parallel for H.O. + threads
Futures and imperative H.O. languages
Context-sensitive analysis on a GPU

Applied
Scripts to programs via analysis
Analysis of the Racket Machine, X10
Contract verification of .5MLOC
vision

Understand higher-order program analysis
vision

Systematic approach that scales
vision

Systematic approach that scales
vision

Tools for reasoning about large-scale software written in expressive, modern languages.
vision
Tools for reasoning about large-scale software written in expressive, modern languages.

Thank you