Secrets of the Universe

The Ultimate Formal Verification Talk

Charles Averill

UTD Computer Security Group
The University of Texas at Dallas

Fall 2023

Table of Contents

Who am I?

- 21 years old
- Undergraduate CS, Physics senior at University of Texas at Dallas
- Researching source-free formal verification of binary programs with Dr. Kevin Hamlen
- Officer of UTD Computer Security Group
- Applying to FV Ph.D. programs
- Teaching an Introduction to Compiler Design course this semester: https://seashell.charles.systems/teaching/ICD
- My dog's name is Beth
- This is my last CSG talk ever <3

The Philosophy of Uncertainty

"*The only true wisdom is in knowing you know nothing.*" - Socrates

What do we know?

- Start simple, things like:
  - `0 = 0`
  - `red + blue = purple`
  - `salt is salty`
- Maybe we know simple algebraic relations:
  - `If x = 0, then x + y = y`
  - $\sqrt{n^2} = n$
- Maybe we've even studied some more complex things:
  - Supply-Demand curves
  - Cell apoptosis
  - General Relativity

Uncertainty

- We really can't know anything for certain
- Our bodies are designed to observe stimuli and respond to them in order to maximize survival rate, any kind of intelligence that developed after that was a side effect
- As a result, we fall into certainty traps all of the time, these are large reasons why
  - Database breaches
  - Physical infrastructure failures
  - Under-performing public policy

  occur
- We have checks to mitigate failures like these, but they clearly are not 100% effective!

Certainty in Security

"*HIC MANEBIMVS OPTIME*" - Marcus Furius Camillus

Hope

- We have established that we live in a low-certainty world
- This is not the end: enter High-Assurance Computing
- "Let's make rigorous, mathematically-defined checkable models of computing so we can verify that we made the software correctly"

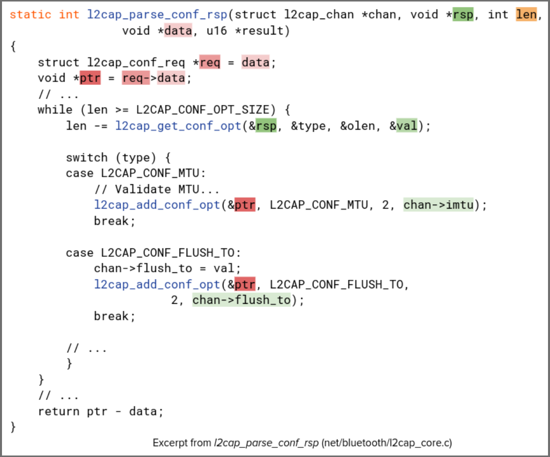
Pros	Cons
Enhanced security	Expensive
High reliability	Takes a long time
Less maintenance	Significantly more difficult
High trustworthiness	Hard to scale

Case Study

- Many examples of “dumb” software bugs with huge impacts
- Heartbleed (2014): Buffer overflow in assumed benign code allows for huge information leakage
- Shellshock (2014): Unjustified trust in environment variables allows for RCE on the majority of online systems
- Spectre/Meltdown (2018): Unjustified trust in information-concealing properties of speculative execution allows for huge information leakage
- BlueBorne (2017): Let’s find out!

BlueBorne

- Collection of 8 cross-platform vulnerabilities in the Bluetooth stack
- One of the Linux vulnerabilities - This one allows for RCE

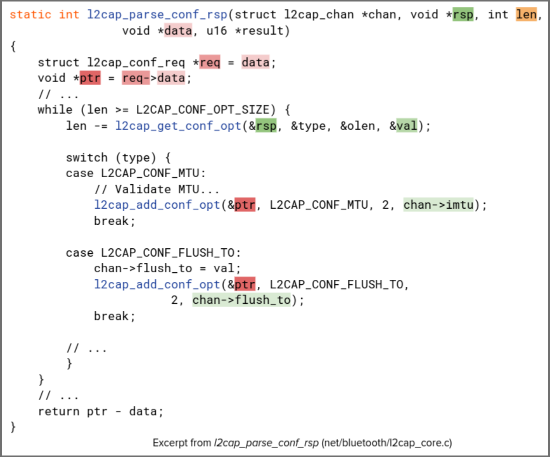


```
static int l2cap_parse_conf_rsp(struct l2cap_chan *chan, void *rsp, int len,
                void *data, u16 *result)
{
    struct l2cap_conf_req *req = data;
    void *ptr = req->data;
    // ...
    while (len >= L2CAP_CONF_OPT_SIZE) {
        len -= l2cap_get_conf_opt(&rsp, &type, &olen, &val);

        switch (type) {
        case L2CAP_CONF_MTU:
            // Validate MTU...
            l2cap_add_conf_opt(&ptr, L2CAP_CONF_MTU, 2, chan->imtu);
            break;

        case L2CAP_CONF_FLUSH_TO:
            chan->flush_to = val;
            l2cap_add_conf_opt(&ptr, L2CAP_CONF_FLUSH_TO,
                       2, chan->flush_to);
            break;

        // ...
        }
    }
    // ...
    return ptr - data;
}
```

Excerpt from *l2cap_parse_conf_rsp* (net/bluetooth/l2cap_core.c)

BlueBorne

`rsp` is an attacker-controlled buffer, `l2cap_parse_conf_rsp` intends to parse `rsp`, validate it, and copy it into `data`. Do you see the issue?

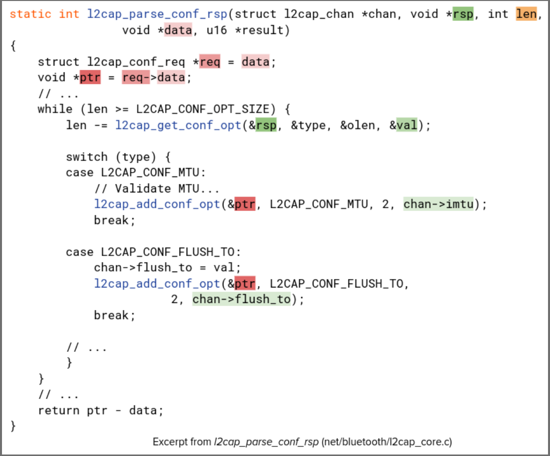


```
static int l2cap_parse_conf_rsp(struct l2cap_chan *chan, void *rsp, int len,
                void *data, u16 *result)
{
    struct l2cap_conf_req *req = data;
    void *ptr = req->data;
    // ...
    while (len >= L2CAP_CONF_OPT_SIZE) {
        len -= l2cap_get_conf_opt(&rsp, &type, &olen, &val);

        switch (type) {
        case L2CAP_CONF_MTU:
            // Validate MTU...
            l2cap_add_conf_opt(&ptr, L2CAP_CONF_MTU, 2, chan->imtu);
            break;

        case L2CAP_CONF_FLUSH_TO:
            chan->flush_to = val;
            l2cap_add_conf_opt(&ptr, L2CAP_CONF_FLUSH_TO,
                        2, chan->flush_to);
            break;

        // ...
        }
    }
    // ...
    return ptr - data;
}
```

Excerpt from *l2cap_parse_conf_rsp* (net/bluetooth/l2cap_core.c)

BlueBorne

The size of the data buffer isn't taken into account! A payload can be crafted in rsp that overflows data and writes arbitrary data into memory.



```
static int l2cap_parse_conf_rsp(struct l2cap_chan *chan, void *rsp, int len,
                void *data, u16 *result)
{
    struct l2cap_conf_req *req = data;
    void *ptr = req->data;
    // ...
    while (len >= L2CAP_CONF_OPT_SIZE) {
        len -= l2cap_get_conf_opt(&rsp, &type, &olen, &val);

        switch (type) {
        case L2CAP_CONF_MTU:
            // Validate MTU...
            l2cap_add_conf_opt(&ptr, L2CAP_CONF_MTU, 2, chan->imtu);
            break;

        case L2CAP_CONF_FLUSH_TO:
            chan->flush_to = val;
            l2cap_add_conf_opt(&ptr, L2CAP_CONF_FLUSH_TO,
                        2, chan->flush_to);
            break;

        // ...
        }
    }
    // ...
    return ptr - data;
}
```

Excerpt from *l2cap_parse_conf_rsp* (net/bluetooth/l2cap_core.c)

BlueBorne

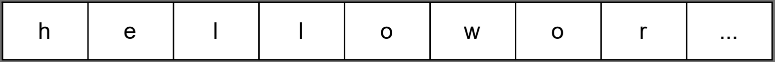
- Can usually be triggered without any user interaction due to some exfiltration techniques overlooked by the Bluetooth specification
- Most BT devices are always listening for traffic directed to them, and only a hardware address is needed to initiate connection
- 5.3b devices at risk at time of discovery, 2b after 1 year
- Hardware address supposed to be difficult to find, but actually fairly easy if you can sniff BT packets due to plenty of un-encrypted header data
- Stack overflows like this one usually mitigated by stack protection techniques - many Linux devices don't use these by default
- If standard mitigations don't work, what does?

Formal Verification of Simple Memory

Memory is a function:

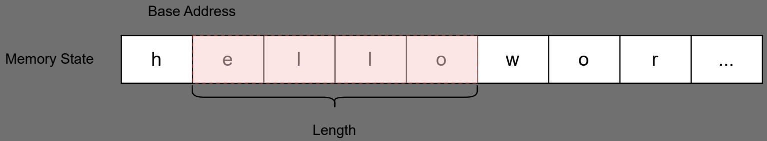
$$mem(x) = \text{contents of memory at position } x$$

$$set(mem, x, data) = \text{new memory state with } data \text{ at position } x$$



Arrays are a high-level construct that live on top of memory:

$$array = \text{memory state} \cup \text{base array address} \cup \text{array length}$$



Formal Verification of Simple Memory

Accessing, writing data in an array:

$$\begin{aligned} &array.mem(array.base\_addr + index) \\ &\qquad set(array.mem, array.base\_addr + index, data) \end{aligned}$$

Is this safe? No!

Proposed safe accesses and writes:

$$\begin{aligned} array.mem(array.base\_addr + (index \bmod array.size) \\ set(array.mem, (array.base\_addr + (index \bmod array.size, data) \end{aligned}$$

Can we prove this? Yes!

Should you trust me/the system? NO! Let's figure out why it works.

Untyped Lambda Calculus

"*Because Schönfinkel has in no way shown how the introduction of the other fundamental concepts is to be avoided, and because he cannot define them from others, he has not justified his claim. In fact he has achieved only a new and inconvenient notation.*" - Haskell Curry

The Philosophy of Computation

- Lambda Calculus is a computing model that uses simple math definitions, instead of mechanical description of Turing Machines
- LC/TM/CC came about during a rough time in mathematics (1890-1930) when paradoxes had been found in our assumptions
- Wanted to make sense of what it meant to do computation, or represent an equation, or decide the truth value of a statement

Implementing the Lambda Calculus

$$\begin{aligned} \text{Syntax:} \quad & e := v \mid \lambda v.e \mid (e_1)(e_2) \\ \text{Semantic(s?):} \quad & \frac{}{(\lambda v.e_1)(e_2) \Downarrow e_1[e_2/v]} \end{aligned}$$

That was...easy?

Maybe not very clear, let's try again

Implementing the Lambda Calculus

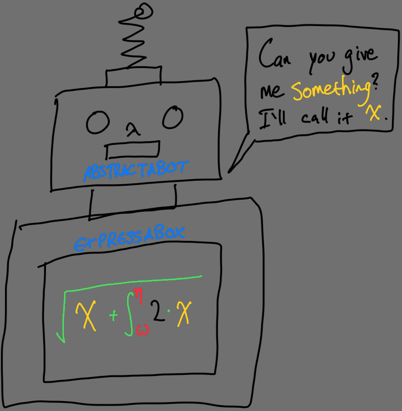
In LC we have "expressions", which can be:

- Variable names (v)
- Functions with single arguments, a.k.a. abstractions ($\lambda v.e$)
- Applications of two other expressions ($(e_1)(e_2)$)

We only need one rule to compute things: when applying two expressions, if the left is an abstraction, take the right expression and plug it into every occurrence of the abstraction's variable in the abstraction's expression.

Implementing the Lambda Calculus

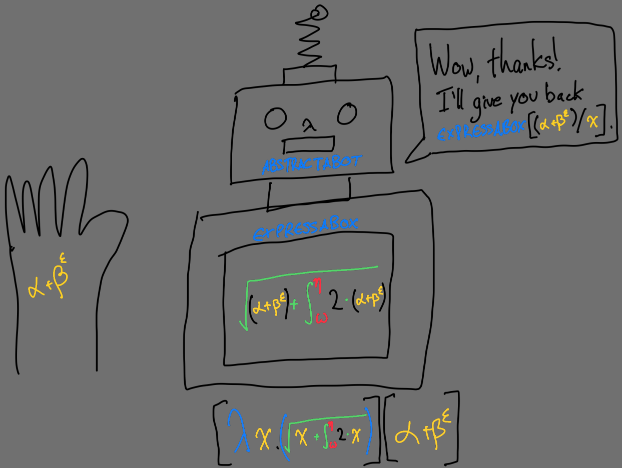
One more explanation for clarity:


λ
ABSTRACTABOT
Can you give me Something? I'll call it x.
EXPRESSABOX
$\sqrt{x + \int_{\omega}^{\eta} 2 \cdot x}$

$$\lambda x.\left(\sqrt{x + \int_{\omega}^{\eta} 2 \cdot x}\right)$$

Implementing the Lambda Calculus

One more explanation for clarity:


Wow, thanks! I'll give you back EXPRESSABOX$[(\alpha+\beta^\varepsilon)/x]$.
ABSTRACTABOT
EXPRESSABOX
$\sqrt{(\alpha+\beta^\varepsilon)+\int_\omega^\eta 2\cdot(\alpha+\beta^\varepsilon)}$
$\alpha+\beta^\varepsilon$
$[\lambda x.(\sqrt{x+\int_\omega^\eta 2\cdot x})][\alpha+\beta^\varepsilon]$

What can we do with LC?

Numbers:

$$0 := \lambda f.\lambda x.x$$
$$Successor := \lambda n.\lambda f.\lambda x.f(nfx)$$

Booleans:

$$True := \lambda x.\lambda y.x$$
$$False := \lambda x.\lambda y.y$$
$$if\ B\ then\ P\ else\ Q := \lambda B.\lambda P.\lambda Q.BPQ$$

Arbitrary loops: (try this for yourself ◡̈)

$$Y := \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

LC Issues

- No data types - ints can be used as bools and etc. and it's still a legal expression
- Non-terminating expressions (huge wrench in the mathematics side, breaks our single semantic rule)
- Partial evaluation is valid - makes debugging very difficult

How are we going to solve this?

Typed Lambda Calculus

“*For any formal system, we can really only understand its precise details after attempting to implement it.*” - Simon Thompson

Simply-Typed Lambda Calculus

As with many things in Computer Science, we can ameliorate our issues with LC via Type Theory.

We are going to add data types to LC and see where it takes us. We will gain things and lose things when we do this!

STLC

$$e := () \mid v \mid \lambda v : t.e \mid (e_1)(e_2)$$
$$t := \texttt{unit} \mid t_1 \to t_2$$

$$(1)\,\frac{}{(\lambda v : t.e_1)(e_2) \Downarrow e_1[e_2/v]}$$

$$(2)\,\frac{}{\texttt{typeof}(()) : \texttt{unit}}$$

$$(3)\,\frac{\texttt{typeof}(v) : t_1 \qquad \texttt{typeof}(e) : t_2}{\texttt{typeof}(\lambda v : t_1.e) : t_1 \to t_2}$$

$$(4)\,\frac{\texttt{typeof}(e_1) : t_1 \to t_2 \qquad \texttt{typeof}(e_2) : t_1}{\texttt{typeof}(e_1 e_2) : t_2}$$

Examples

$$\lambda x : \texttt{int}.\ x + 5$$

$$\lambda s : \texttt{string}.s\ \texttt{++}\ \text{``hello world''}$$

$$\lambda f : (\texttt{int} \to \texttt{int}).\lambda g : (\texttt{int} \to \texttt{int}).\lambda n : \texttt{int}.f(g(n))$$

Type Inhabitation

Another Theoretical Thing

- Let's take what seems to be a detour and think about a fun little puzzle.
- First, we will add a new type to our STLC: `void`. `void` is special, because there is no value that has type void. Therefore, the type `void` is uninhabited.
- We know due to rule (2) that the expression () has type `unit`, so we say that "`unit` is inhabited by the value ()."
- We can show that the type `unit` → `unit` is inhabited by the value:

$$\lambda x : \texttt{unit}.x$$

Another Theoretical Thing

So if `void` is uninhabited, and `unit` is inhabited, and $\texttt{unit} \to \texttt{unit}$ is inhabited, is this type inhabited? Yes! $\lambda x : \texttt{void}.()$

$$\texttt{void} \to \texttt{unit}$$

What about this type? Yes! $\lambda x : \texttt{void}.x$

$$\texttt{void} \to \texttt{void}$$

What about this type? No!

$$\texttt{unit} \to \texttt{void}$$

More Types

Very quickly, let's add some more types:

- **Pairs**: (e_1, e_2). These expressions have type $\texttt{typeof}(e_1) * \texttt{typeof}(e_2)$, we call them "product types"
- **Constructed Types**: $\texttt{CON1}^{t_1+t_2}(e) \mid \texttt{CON2}^{t_1+t_2}(e)$. These expressions have type $t_1 + t_2$, we call them "sum types"

We can't make a pair that has a void in it, because no expression has type `void`.

But, can we make a sum type with the signature $\texttt{unit} + \texttt{void}$?

Yes! One value of this type is $\texttt{CON1}^{\texttt{unit}+\texttt{void}}(())$

Try showing that the following is inhabited:

$$\texttt{unit} * (\texttt{unit} \to (\texttt{void} + (\texttt{unit} \to \texttt{unit})))$$

Question Intermission

Binding Types with Logic

Type Checking

- Remember that we had some rules about the valid types of STLC expressions

$$(2)\frac{}{\texttt{typeof}(()) : \texttt{unit}}$$

$$(3)\frac{\texttt{typeof}(v) : t_1 \qquad \texttt{typeof}(e) : t_2}{\texttt{typeof}(\lambda v : t_1 . e) : t_1 \to t_2}$$

$$(4)\frac{\texttt{typeof}(e_1) : t_1 \to t_2 \qquad \texttt{typeof}(e_2) : t_1}{\texttt{typeof}(e_1 e_2) : t_2}$$

- The whole point of these is to be able to **statically** check the program to ensure that it's well-typed (meaning we can't use numbers as booleans or functions or etc.)
- Let's see how a simple type checker works

Curry-Howard Isomorphism

We're finally ready to assemble all of these pieces into (in my opinion) the most beautiful mathematical theory ever discovered: The Curry-Howard Isomorphism.

In short, it states that types are theorems, and programs are proofs of those theorems. Let's dig into why.

Why we did Type Inhabitation

The kinds of types we added to STLC were very deliberate:

Type	Inhabited
unit	True
void	False
void * void	False
void * unit	False
unit * void	False
unit * unit	True
void + void	False
void + unit	True
unit + void	True
unit + unit	True

Type	Inhabited
void → void	True
void → unit	True
unit → void	False
unit → unit	True

Why we did Type Inhabitation

The kinds of types we added to STLC were very deliberate:

Logical Expression	Truth Value
T	True
F	False
$F \wedge F$	False
$F \wedge T$	False
$T \wedge F$	False
$T \wedge T$	True
$F \vee F$	False
$F \vee T$	True
$T \vee F$	True
$T \vee T$	True

Logical Expression	Truth Value
$F \to F$	True
$F \to T$	True
$T \to F$	False
$T \to T$	True

Why the CHI matters

Think about that - we have

- A language that natively encodes the philosophical ideas of theorems and proofs within its types and expressions
- That can be (somewhat) trivially type-checked
- Its semantics are simple enough that they fit on a single presentation slide

That means that with careful engineering, and lots of confidence from lots of mathematicians, we can create a

high-assurance automated proof checker.

Enter Coq.

Coq

"*Logic takes care of itself; all we have to do is to look and see how it does it.*" - Ludwig Wittgenstein

Coq

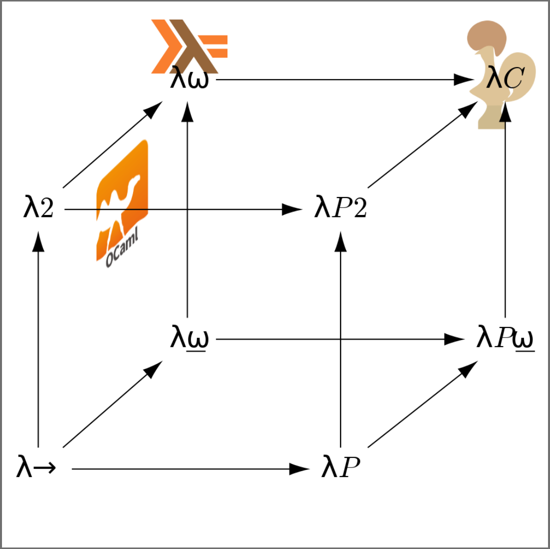
- Coq is an automated theorem proving system, containing a programming language called Gallina, as well as a proof language
- Coq is essentially an **implementation** of the Curry-Howard isomorphism, binding the concepts of types, theorems, programs, and proofs into a cohesive lambda calculus (CiC) that allows for high-assurance proof checking
- Has an extremely small TCB hand-verified by thousands of mathematicians for decades, and now machine-checked by the MetaCoq project

The Calculus of Inductive Constructions

Coq relies on the Calculus of Inductive Constructions (CiC or λC), a variant of typed lambda calculus that combines the three primary type system additions, to provide the expressiveness necessary to state theorems that we're interested in:

- Parametric Polymorphism (type to term) - Adds a new kind of abstraction that takes a **type** as input and returns an expression ($\Lambda\alpha.e$) - allows us to express properties of generic data structures
- Dependent Types (term to type) - Adds the capability to define expressions with types that change depending on the contents of the expression (e.g. `int list 5` vs `int list 3`) - this gives us the ability to use function "calls" as components in our theorems, and the $\forall$ operator
- Type Constructors (type to type) - Adds a new kind of abstraction that takes a **type** as input and returns a new type ($\Pi\alpha.t$) - this is necessary to avoid some paradoxes about the "type of types", also gives us the $\exists$ operator

Lambda Cube



$$\begin{tikzcd}
 & \lambda\omega \arrow[rr] & & \lambda C \\
\lambda 2 \arrow[ur] \arrow[rr] & & \lambda P2 \arrow[ur] & \\
 & \lambda\underline{\omega} \arrow[uu] \arrow[rr] & & \lambda P\underline{\omega} \arrow[uu] \\
\lambda\to \arrow[uu] \arrow[ur] \arrow[rr] & & \lambda P \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

A Proof

```
Inductive nat : Type :=
| O
| S (n : nat).

Theorem add_0_r:
  forall (n : nat), n + 0 = n.
Proof.
  intros. induction n.
  (* if n = 0 *)
  - reflexivity.
  (* if n = S n' *)
  - simpl. rewrite IHn. reflexivity.
Qed.

Theorem andb_true: forall (b : bool), b && true = b.
Proof. intros. destruct b; reflexivity. Qed.
```

Breakdown

```
Inductive nat : Type :=
| O
| S (n : nat).
```

"There is a thing called 'nat' and it can either be O or it can be S applied to another nat."

Breakdown

```
Theorem add_0_r:
  forall (n : nat), n + 0 = n.
Proof.
  intros. induction n.
  (* if n = 0 *)
  - reflexivity.
  (* if n = S n' *)
  - simpl. rewrite IHn. reflexivity.
Qed.
```

"I propose this thing called 'add_0_r' which says that for all natural numbers n, $n + 0 = n$. I will prove it via an induction on n, using the inductive hypothesis in the inductive step."

Breakdown

```
Theorem andb_true:
  forall (b : bool), b && true = b.
Proof.
  intros. destruct b; reflexivity.
Qed.
```

"I propose this thing called 'andb_true' which says that for all booleans b, b && true = true. I will prove it via a case analysis of b."

$$\text{Me: "I think this type is inhabited:"}$$

$$\texttt{bool} : b \to (\texttt{eq}\ \ (\texttt{andb}\ \ b\ \ \texttt{true})\ b)$$

$$\text{Coq: "I don't believe you"}$$

$$\text{Me: "I'll show you it's inhabited:"}$$

$$\lambda\, b : \texttt{bool}.\ \ \texttt{case}\ \ b\ \ \texttt{of}$$

$$\mid \texttt{false} \to \texttt{eq\_refl}\ \ \texttt{(andb false true)}\ \ \texttt{false}$$

$$\mid \texttt{true} \to \texttt{eq\_refl}\ \ \texttt{(andb true true)}\ \ \texttt{true}$$