HALO: Haskell to Logic through Denotational Semantics

Dimitrios Vytiniotis, Simon Peyton Jones,
Koen Claessen, Dan Rosén

Submitted to POPL 2013

`web.student.chalmers.se/~danr/halo-popl.pdf`

Static Contract Checking for Haskell

```
risers []  = []
risers [x] = [[x]]
risers (x:y:ys)
  = case risers (y:ys) of
       [] -> error "Crash"
       (s:ss) -> if x <= y then (x:s):ss
                           else [x]:s:ss
```

CAN THIS CODE CRASH?

```
risers ∈ CF && {xs | not (null xs)} ->
          CF && {ys | not (null ys)}
```

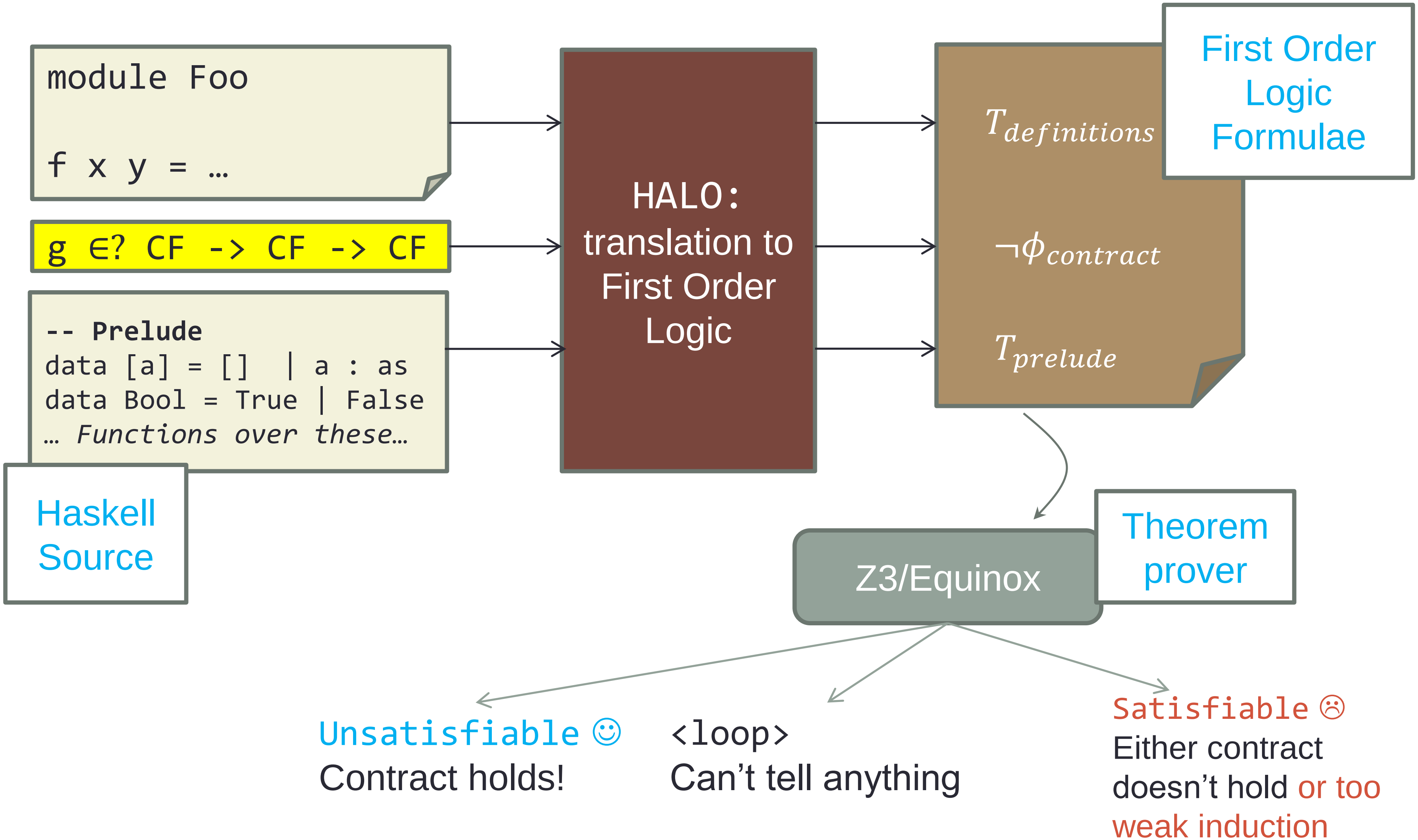
Syntax of contracts

```
C ::= {x|p} | (x:C) -> C | C && C | CF
```

Just a Haskell expression of type Bool

"crash-free"

Check contracts using an FOL theorem prover



Satisfying a contract, denotationally

- What does it really mean to satisfy a contract?

$$\boxed{[\![\mathtt{C}]\!]_\rho \subseteq D_\infty}$$

$$[\![x \mid e]\!]_\rho = \{d \mid d = \bot \vee [\![e]\!]_{\rho,x\mapsto d} \in \{\mathsf{True}, \bot\}\}$$

$$[\![(x{:}\mathtt{C}_1) \to \mathtt{C}_2]\!]_\rho = \{d \mid \forall d' \in [\![\mathtt{C}_1]\!]_\rho \,.\, \mathsf{app}(d, d') \in [\![\mathtt{C}_2]\!]_{\rho,x\mapsto d'}\}$$

$$[\![\mathtt{C}_1 \& \mathtt{C}_2]\!]_\rho = \{d \mid d \in [\![\mathtt{C}_1]\!]_\rho \wedge d \in [\![\mathtt{C}_2]\!]_\rho\}$$

$$[\![\mathtt{CF}]\!]_\rho = F^\infty_{\mathsf{cf}}$$

- Define contract satisfication: `f ∈ C` to be [f] ∈ [C]

Example: Base contracts to FOL

(Part of $\phi_{contract}$)

C[e ∊ C] = translation of "expression e satisfies contract C"

- Predication: using a function `p : a -> Bool`

```
C[e ∈ {x | p}] ≔
         E[e] = UNR
    ∨    E[p[e/x]] = True
    ∨    E[p[e/x]] = UNR
```

- Crash-freeness:

```
C[e ∈ CF] ≔ CF(E[e])
```

where `CF` axiomatises domain-theoretic crash-freedom

Justified by Denotational Semantics

Theorem 4.6. *Assume that e and $\mathtt{C}$ contain no free term variables. Then the FOL translation of the claim $e \in \mathtt{C}$ holds in the model if and only if the denotation of e is in the semantics of $\mathtt{C}$. Formally:*

$$\langle D_\infty, \mathcal{I} \rangle \models \mathcal{C}\{\!\{e \in \mathtt{C}\}\!\} \quad \Leftrightarrow \quad [\![e]\!] \in [\![\mathtt{C}]\!]$$

- Say Z3 proves $T_{definitions}, T_{prelude}, \neg\phi_{contract}$ UNSATISFIABLE
- What does it say about the contract in our program?

- Using a uni-typed denotational model D^∞, we show:

$$D^\infty \models T_{definitions}, T_{prelude}$$

- Hence a proof as above gives us:

$$D^\infty \models \phi_{contract}$$

i.e. the contract holds for the program.

We have a tool that implements this!

Description	equinox	Z3	vampire	E
ack CF	-	0.04	0.03	-
all CF	-	0.00	3.36	0.04
(++) CF	-	0.03	3.30	0.38
concatMap CF	-	0.03	6.60	-
length CF	0.87	0.00	0.80	0.01
(+) CF	44.33	0.00	3.32	0.10
(*) CF	6.44	0.03	3.36	-
factorial CF	6.69	0.02	4.18	31.04
exp CF	-	0.03	3.36	-
(*) accum CF	-	0.03	3.32	-
exp accum CF	-	0.04	4.20	0.12
factorial accum CF	-	0.03	3.32	-
reverse CF	13.40	0.03	28.77	-
(++)/any morphism	-	0.03	-	-
filter satisfies all	-	0.03	-	-
iterate CF	5.54	0.00	0.00	0.00
repeat CF	0.06	0.00	0.00	0.01
foldr1	-	0.01	1.04	24.78
head	18.62	0.00	0.00	0.01
fromJust	0.05	0.00	0.00	0.00
risersBy	-	-	1.53	-
shrink	-	0.04	-	-
withMany CF	-	0.00	-	-

Figure 7: Theorem prover running time in seconds on some of the problems in the test suite on contracts that hold.

BUT
if the contract doesn't hold
all theorem provers
diverge!

Backup slides

- :)

Partial applications

- Axioms for partial applications

```
f :: Int -> Int -> Bool
f x y = …
```

  - $\forall$ x y, f(x,y) = e
  - $\forall$ x y, app(app(f_ptr,x),y) = f(x,y)

  - Treat partial applications by using f_ptr and app() instead of f

Moving forward to new territory

- Enable finite counter models for contracts that don't hold: users can then get counterexamples
  - Can we use our ideas using triggers in Z3?
  - These heuristics can guide theorem provers to faster successes

- Richer contract constructs
  - parameterised, partially applied contracts, access to FOL equality

- Wider Haskell coverage
  - Type classes, primitive theories for data types as `Integer`

`web.student.chalmers.se/~danr/halo-popl.pdf`

Fixed point induction

- Replace recursive calls to a fresh function **risers_rec**:

```
risers []  = []
risers [x] = [[x]]
risers (x:y:ys)
    = if x <= y then (x:s):ss else [x]:s:ss
  where (s:ss) = risers_rec (y:ys)
```

- Assume contract holds for the recursive call:

```
Hypothesis:
  risers_rec ∈ CF && {xs | not (null xs)} ->
               CF && {ys | not (null ys)}

To show:
  risers ∈ CF && {xs | not (null xs)} ->
           CF && {ys | not (null ys)}
```

Fixed point induction **only** applicable on admissible predicates! Contracts are!

Static verification for functional programs

Liquid Types [Jhala et al]
- Symbolic contracts
- Predicate abstraction
- Inference
- Strict semantics

ESC/Haskell [Xu et al]
- Contracts are programs
- Symbolic execution/inlining
- Lazy semantics

Zeno [Sonnex et al]
- Automated equality proofs
- Clever heuristics
- Strict semantics

ACL2

Catch [Mitchell]
- Detect pattern match failures
- Via static analysis
- For Haskell

Dafny & Boogie [Leino et al]

Leon [Suter et al]
- Satisfiability mod CBV recursive programs
- Integrated with Scala

2011

Recursive predicates [Bjørner et al]
- Recursive logic programs + SAT

2012

F7/F* [Swamy et al]
- Value-dependent types
- Symbolic predicates

Active research on verification of "pure" recursive programs

Admissibility and induction

- If a predicate is true for all elements of a chain, then it is true for the limit. Not all predicates are admissible
- Our language of contracts *is* admissible:
  - Informal argument: Haskell functions are continuous

$$\frac{admissible(P) \quad P(\mathrm{UNR}) \quad P(h) \Longrightarrow P(F\ h)}{P(fix\ F)}\ [\mathrm{FixInd}]$$

[**Design Principle**]

All predicates of the form

P(f) = f ∈ (`C1 -> … -> Cn -> C`)

are admissible

Minimize to the terms I'm interested in

Lots of quantified axioms in the assumptions, eg:

$$\forall x,\ CF(x) \Longrightarrow CF(f(x))$$

$$\forall x,\ f(x) = E[e]$$

☹ Thm prover instantiating quant. assumptions all the time

☹ Generating new terms all the time

How can we restrict to the terms "I am interested in?"

Solution: introduce a min() predicate guard (like a "trigger")

$$\forall x,\ min(f(x)) \Longrightarrow CF(x) \Longrightarrow CF(f(x))$$

$$\forall x,\ min(f(x)) \Longrightarrow f(x) = E[e]$$

Split contract translation to "assumption" vs. "goal" mode

Minimize to the terms I'm interested in

min() predicate propagates along evaluation, contract translation splits to assumption/goal modes

D[`let` f x = e] = ∀x, **min(f(x))** ⟹ f(x) = E[e]

D[`let` f x = `case` e `of` { K1 `->` e1; K2 `->` e2 }]
= ∀x, **min(f(x))** ⟹
min(E[e])∧((E[e] = K1 ∧ f(x) = E[e1])∨(E[e] = K2 ∧ f(x) = E[e2])
∨ (E[e] = BAD ∧ f(x) = BAD) ∨ (f(x) = UNR))

min(app(e1,e2)) ⟹ **min(e1)**

Type classes: the problem

Do not want to treat dictionaries as records of functions!

```
f :: Eq a => a -> a -> Bool                    -- In source
f x y = x == y

f :: forall a. Eq a => a -> a -> Bool          -- In FC
f = /\a.\dEq x y. (==)@a dEq x y
```

How to show that CF(f dict ‘a’ ‘b’)?

- Only way is by showing that CF(dict); amounts to showing that CF((==)). But the only way to prove this is by axiom:

$$(\forall x, CF(g(x))) \Longrightarrow CF(g)$$

Which can easily make the theorem prover search blow up …

Type classes: solution

Treat type classes as “open GADTs”

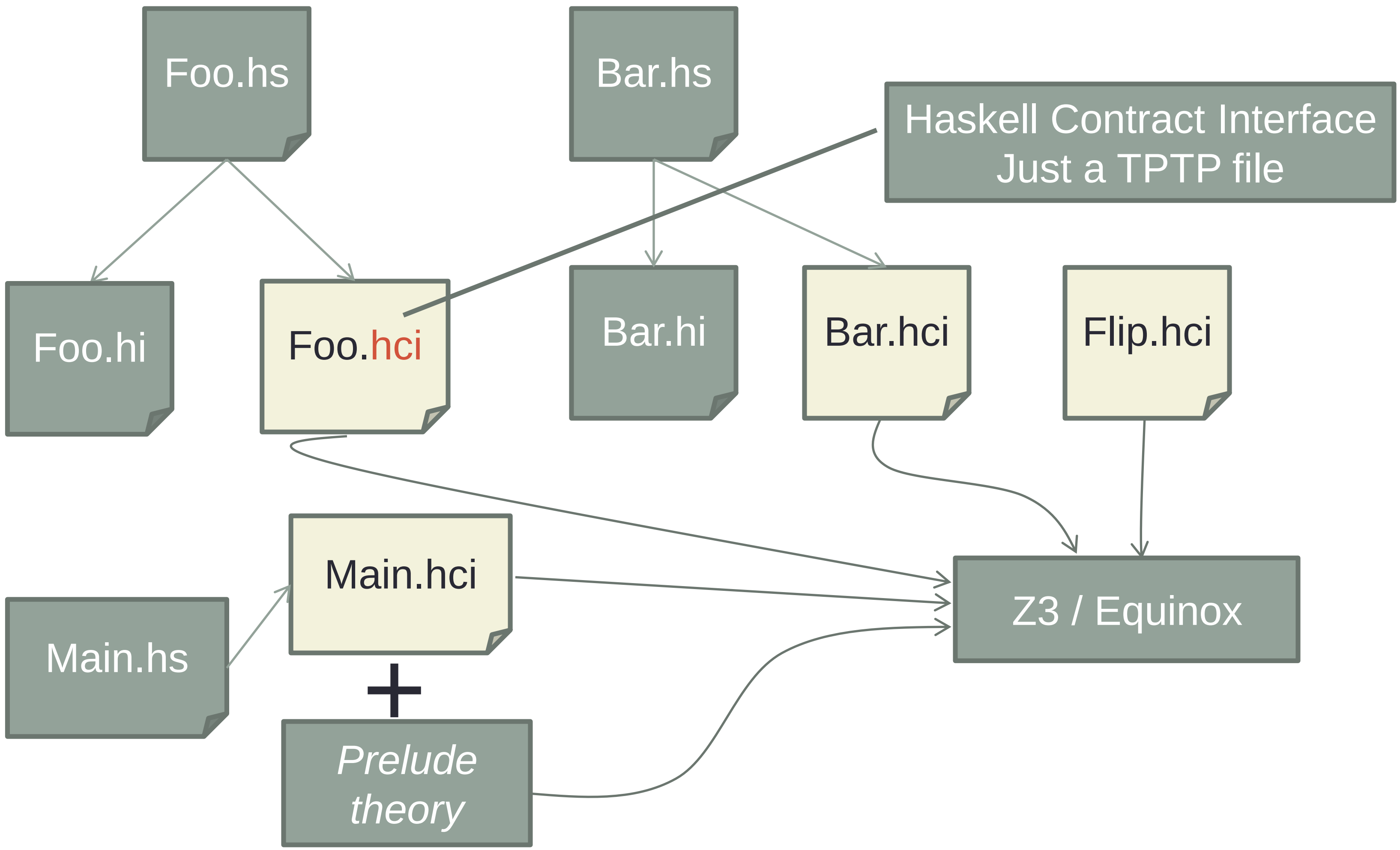
```
data Eq a where
  TInt  :: Eq Int           -- there is an instance Eq Int
  TBool :: Eq Bool          -- there is an instance Eq Bool
  TList :: Eq a -> Eq [a] -- there is an instance Eq a => Eq [a]

(==) :: Eq a => a -> a -> Bool
(==) TInt  = ...
(==) TBool = ...
(==) TList [] [] = True
(==) (TList t) (x:xs) (y:ys)= (==) t x y && (==) (TList t) xs ys
(==) (TList t) _ _ = False
```

Each instance declaration introduces:

- A new constructor with appropriate type
- A new FOL clause that matches on the new constructor

Ideas for separate compilation



Int vs Int#

- In Haskell Int is something like:

```
data Int where
     Int# : Int# -> Int
```

- This means we can treat it as any other datatype and treat the Int# argument as primitive integer, with operations as +# directly interpreted in the theory

- Need FOL + theory of arithmetic