Intrinsically-typed and well-scoped SMT-LIB FFI bindings with modular abstract syntax trees (MAST)

Kajetan Granops, Mihail-Codrin Iftode, and Ohad Kammar

Slides:

2–5 December 2025

IFIP WG2.11: Program Generation

Stellenbosch Institute for Advanced Study, South Africa

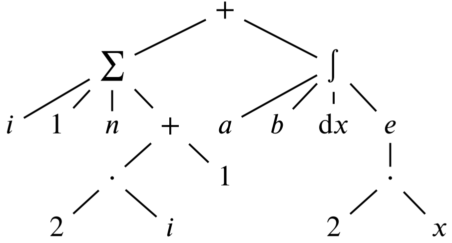
THE UNIVERSITY of EDINBURGH informatics lfcs Laboratory for Foundations of Computer Science BayesCentre Funded by: THE ROYAL SOCIETY Advanced Research + Invention Agency ARIA

$$\left(\sum_{i=1}^{n}(2i+1)\right)+\int_a^b e^{ax}\mathrm{d}x$$

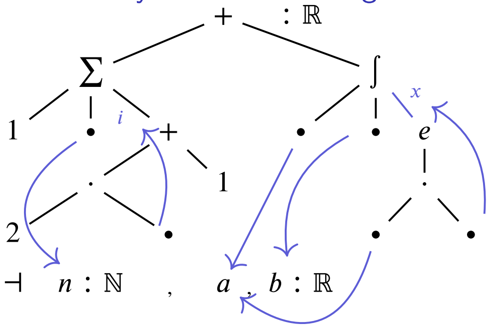
Concrete syntax

“(”, “$\sum$”, “_”, “{”, “i”, “=”, “1”, “}”, “{”, “n”, “(”, “2”, “i”, “+”, “1”, “)”, “}”, …

Abstract syntax



Abstract syntax with binding



Standardised query language for SMT solvers such as Z3 [de Moura-Bjørner'08]:

```
; Integer arithmetic
(set-logic QF_LIA)
(declare-const x Int)
(declare-const y Int)
(assert (= (- x y)
           (+ x (- y) 1)))
(check-sat)
; unsat
(exit)
```

$\xmapsto{\text{z3}}$ `unsat`

Standardised query language for SMT solvers such as Z3 [de Moura-Bjørner'08]:

```
; Integer arithmetic
(set-logic QF_LIA)
(declare-const x Int)
(declare-const y Int)
(assert (= (- x y)
           (+ x 1 (- y) (- 1))))
(check-sat)
(get-value (x y))
(exit)
```

$\xmapsto{\text{z3}}$

```
sat
((x 0)
 (y 0))
```

Satyr: Idris FFI for SMT-LIB (WiP)

Want Z3 queries FFI, e.g.:

- Python's z3py
- Haskell's SBV
- Agda's Schmitty
- ...

Work in progress:

Idris Satyr Intrinsically-typed well-scoped FFI with holes and modular serialisation and deserialisation

Intrinsically-typed well-scoped syntax with splicing[1]

```
query : Term BoolSMT [< "x" :- IntSMT, "y" :- IntSMT]
query = (V "x" - V "y") .=. (V "x" + 1 + (- V "y") + (- 1))
```

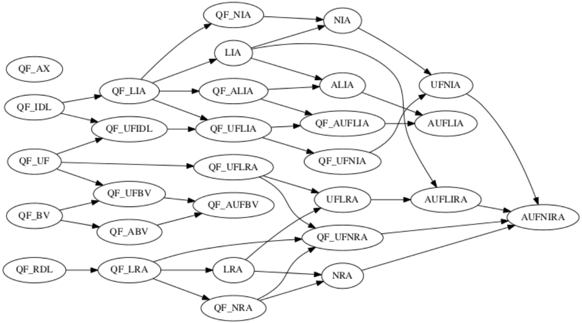
- No dangling variables
- Only represents well-typed queries
- Hole support (not shown)

[1] Illustration purposes only

Need for Modularity

SMT-LIB query language

- S-expressions
- 29 theories
- multiple syntax extensions



Goal

FFI

- Intrinsically-typed well-scoped FFI with holes
- Modular serialisation
- Modular well-scoped parsing
- Modular type-inference

Talk structure

- Problem statement and goal
- Architecture
- Serialisation
- Parsing
- Type reconstruction

Expression problem [Reynolds'75, Cook'90, Krishnamurthi, Felleisen and Friedman'98, Wadler'98]

Spec [Wadler'98]

Both:

- **Extend** object-language syntax
- **Add** meta-language functions/properties of programs

But:

- Without recompiling previous modules; alternatively
- Retaining and reusing both old and new languages

Some solutions

- Scala Mix-ins [Zenger'98, Zenger and Odersky'01]
- Visitor Pattern in Pizza, Zodiac [Krishnamurthi, Felleisen and Friedman'98]
- Recursive Generics [Wadler'98]
- Data-types á la carte: coproducts of signature functors [Swierstra'08]

Data-types á la carte [Swierstra'08]

Initial algebra semantics [Goguen and Thatcher'74]

Abstract syntax (without binding) in Haskell

(Idris implementation on next slide)

Terms á la carte in Idris — Ask me to go through this slide slowly

```
Signature : Type
Signature = Type -> Type

BinOp : Signature
BinOp x = (x,x)

(+) : (x,y : Signature) -> Signature
(s + t) x = Either (s x) (t x)

IntSig : Signature
IntSig = BinOp + BinOp + const Integer

data (.Term) : Signature -> Type -> Type where
  V  : a -> sig.Term a
  Op : sig (sig.Term a) -> sig.Term a
```

```
Num (IntSig .Term x) where
  x + y = Op (Left (Left  (x,y)))
  x * y = Op (Left (Right (x,y)))
  fromInteger x = Op (Right x)

ex1 : IntSig .Term String
ex1 = V "x" + V "y"
```

Terms á la carte in Idris

```
Signature : Type
Signature = Type -> Type

BinOp : Signature
BinOp x = (x,x)

(+) : (x,y : Signature) -> Signature
(s + t) x = Either (s x) (t x)

IntSig : Signature
IntSig = BinOp + BinOp + const Integer

data (.Term) : Signature -> Type -> Type where
  V  : a -> sig.Term a
  Op : sig (sig.Term a) -> sig.Term a
```

```
Num (IntSig .Term x) where
  x + y = Op (Left (Left  (x,y)))
  x * y = Op (Left (Right (x,y)))
  fromInteger x = Op (Right x)

ex1 : IntSig .Term String
ex1 = V "x" + V "y"
```

Evaluator á la carte in Idris

```
(.AlgebraOver) : Signature -> Type -> Type
sig.AlgebraOver a = sig a -> a

plus : BinOp .AlgebraOver Integer
plus (x,y) = x + y

times : BinOp .AlgebraOver Integer
times (x,y) = x * y

constant : (const a).AlgebraOver a
constant x = x

(:+:) : s.AlgebraOver a -> t.AlgebraOver a -> (s+t).AlgebraOver a
(salg :+: talg) (Left  sVal) = salg sVal
(salg :+: talg) (Right tVal) = talg tVal

intAlg : IntSig .AlgebraOver Integer
intAlg = ((plus :+: times)    {s = BinOp, t = BinOp}
                :+: constant) {s = BinOp + BinOp, t = const Integer}
```

Data-types á la carte [Swierstra'08]

Initial algebra semantics [Goguen and Thatcher'74]

Abstract syntax (without binding) in Haskell

Coq á la carte [Forster and Stark'20]

Abstract syntax with binding through metaprogramming.

We want:

Abstract syntax á la carte without metaprogramming.

Mathematical theory for AST á la carte: SOAS [Fiore, Plotkin, and Turi '99]

Second-Order Abstract Syntax (SOAS)

- Initial algebra characterisation for
  abstract syntax with binding-aware substitution
- Robust to extensions:
  - polymorphism [Fiore and Hamana'13]
  - mechanisation [Crole'11, Allais et al.'18, Fiore and Szamozvancev'22]
  - substructurality [Fiore and Ranchod'25]
- CBN works smoothly. Doesn't cover CBV for technical reasons:
  - Substitute **in**: values and terms
  - Substitute for variables: values only

Required extension: sort classification

- We substitute **in** all sorts
- Support variables and substitute for 1^{st}-class only.

Mathematical theory: MAST [Fiore, Kammar, Moser and Staton'25]

Modular Abstract Syntax Trees (MAST)

- SOAS $\overset{\text{generalise}}{\rightsquigarrow}$ 2^{nd}-class sorts
  - Kleisli bicategories [Gambino, Fiore, Hyland, and Winskel'19]
  - Generalise monoidal categories to actegories
  - Generalise substitution monoids to actions / modules
- Case-study: CBV semantics á la carte (128 substitution lemmata)

Takeaway (modularity)

Each syntactic construct defines its own binding, renaming, and substitution structure. Combine syntax á la carte with generic traversals.

MAST paper

This talk

- Computational implementation
- Application to the Expression Problem

Talk structure

- Problem statement and goal
- Architecture
- Serialisation
- Parsing
- Type reconstruction

Satyr Architecture

$$
\begin{tikzcd}
 & \boxed{\text{User-facing GADT (hand-written, non-modular)}} \arrow[d, shift left, "\text{conversion functions (hand-written)}"] & \\
 & \boxed{\text{MAST IR (modular)}} \arrow[u, shift left] \arrow[dl, "\text{serialiase (modular, generic)}"'] & \\
\boxed{\text{SMT-LIB source}} \arrow[rr, "\text{z3}"] & & \boxed{\text{SMT-LIB source}} \arrow[ul, "\text{deserialise (modular, generic)}"']
\end{tikzcd}
$$

User-facing interface[2]

```
data Term : TypeSMT -> Context -> Type where
  (.=.) : Term a ctx -> Term a ctx
       -> Term BoolSMT ctx

  (-),(+),(*) : Term IntSMT ctx -> Term IntSMT ctx
              -> Term IntSMT ctx

  (.-) :    Term IntSMT ctx
        -> Term IntSMT ctx

  Var   : (pos : (s :- ty) `Elem` ctx)
      -> Term ty ctx
  Const : Integer -> Term IntSMT ctx
```

[2]Illustration purposes only.

MAST signature functors

Building blocks: signature combinators

- $(\looparrowright_t)$ Extension: define, e.g., only an integer operator
- $(@t)$ Application: project, e.g., only an integer sub-term
- $(\Gamma\rhd)$ Context shift: bind variables in subterm
- $(\coprod, \prod)$ Sums and products: as in á la carte

Binding signatures [Aczel'78]

$$\left([\Gamma_1.s_1, \ldots, \Gamma_n.s_n] \Rightarrow t\right) \coloneqq \underset{t}{\looparrowright} \prod_{i=1}^{n} \Gamma_i \rhd - @\ s_i$$

Architecture: query[3]

q : `Term BoolSMT ctx` | r : `Env ctx`

convert (-written, non-modular) ↓ | ↑map convert (non-modular)

qIR : `TermIR BoolSMT ctx` | rIR : `EnvIR ctx`

serialise (modular) ↓ | ↑type reconstruction (modular)

query : `String` | rRaw : `EnvRaw ctx`

call z3↘ | ↗scoped parser (modular)

sat $((x_0\ v_0)\ (x_1\ v_1)\ \ldots\ (x_n\ v_n))$

[3]Illustration purposes only.

Talk structure

- Problem statement and goal
- Architecture
- Serialisation
- Parsing
- Type reconstruction

MAST traversals[4]

`fold alg ren subst var metavarEnv : Term tySMT ctx → b tySMT ctx`

- `b`: SMT-LIB-type-indexed, context-indexed family
- `alg`: algebra, i.e., per-construct function
- `ren`: how to rename variables in the algebra
- `subst`: how to substitute in the algebra
- `metavarEnv`: what to splice into holes (if any)

[4] Illustration purposes only

Serialisation via MAST traversal

- Serialise with user-preferred names.
- SMT-LIB is nominal, MAST is de Bruijn. Serialisation avoids capture if needed via mangling.

  Open problem(?): nominal syntax without shadowing avoiding per-name de Bruijn?
- Renaming permutes the preferred-name dictionary
- Substitution serialises to object-level `let`-binding.

Architecture: query[3]

q : `Term BoolSMT ctx`

convert (-written, non-modular) ↓

qIR : `TermIR BoolSMT ctx`

serialise (modular) ↓

query : `String`

call Z3 ↘

sat $((x_0\ v_0)\ (x_1\ v_1)\ \ldots\ (x_n\ v_n))$

↗ scoped parser (modular)

rRaw : `EnvRaw ctx`

↑ type reconstruction (modular)

rIR : `EnvIR ctx`

↑ map convert (non-modular)

r : `Env ctx`

[3] Illustration purposes only.

Talk structure

- Problem statement and goal
- Architecture
- Serialisation
- Parsing
- Type reconstruction

TAP: linear-time parser-combinators [Krishnaswami and Yallop'19]

Typed algebraic approach to parsing

Linear time parser combinator library through:

- μ-regexes: regexes + guarded fixpoints
- restricted by static analysis that ensures:
  - grammatical disambiguity
  - 1-lookahead recursive descent parsability

Dependently-typed extension [Greg Brown]

- Extend parser with dependently-typed state.
- Parsers can emit well-scoped raw terms.
- Per-construct parser fragments
- Combined to per-theory parsers.

Talk structure

- Problem statement and goal
- Architecture
- Serialisation
- Parsing
- Type reconstruction

Type reconstruction

Simple constraint solving

- Relatively straightforward.
- Reconstruct types per-construct.
- Per-construct solver fragments.
- Combined to per-theory solver.

Summary

Idris Satyr Intrinsically-typed well-scoped FFI with holes and modular serialisation and deserialisation

q : `Term BoolSMT ctx` | r : `Env ctx`

convert (-written, non-modular) ↓ | ↑map convert (non-modular)

qIR : `TermIR BoolSMT ctx` | rIR : `EnvIR ctx`

serialise (modular) ↓ | ↑type reconstruction (modular)

query : `String` | rRaw : `EnvRaw ctx`

call z3↘ | ↗scoped parser (modular)

sat $((x_0\ v_0)\ (x_1\ v_1)\ \ldots\ (x_n\ v_n))$