# Building a Haskell Verifier out of component theories

Dick Kieburtz
WG2.8, Frauenchiemsee, June 2009

# Why a verifier for Haskell, in particular?

- Feasibility:
  - There's a recognized, stable version that is pretty well defined
    - *Haskell 98*
      - Mature compilers and interpreters exist
      - A collection of papers specifies nearly all aspects of its semantics denotationally
        - a modular, categorical semantics for datatypes provides an equational theory for the operations of each type
    - A programming logic has been developed -- *P-logic*
      - *P-logic* refines the Haskell 98 type system
        - properties of functions are stated as dependent types
      - it takes advantage of the referential transparency of the Haskell language
    - A front-end processor (*pfe*) comprehends both language and logic
- Challenges:
  - *Haskell 98* is a rich language
    - Embodies both lazy and strict semantics
    - Higher-order function types
    - Recursion in both expression and type definitions

# What's new?

After experimenting with the construction of an *ad hoc* verifier (*Plover*) for two years, it became unmaintainable; a new approach was called for.

- I needed an architecture that was modular, provably sound, and could be developed incrementally

- DPT to the rescue!
  - DPT (Decision Procedure Toolkit) is an open-source toolkit for integrating decision procedures with a first-order satisfiability solver
    - Written in OCAML by a team of researchers at Intel
      - (Jim Grundy, Amit Goel, Sava Krstic)
    - Gives state-of-the-art performance
    - The decision-procedure integration strategy is based upon ten simple rules and has been proved sound (Krstic & Goel, 2007)
    - Distributed via Sourceforge

But how can a solver for decidable, first-order logic formulas be used to verify properties of Haskell programs?

# Components of a complex theory are its subtheories

- Let’s take the semantic theory of Haskell 98, for example
  - Subtheories include:
    - Equality
    - Uninterpreted functions
    - Cartesian products
    - Definedness of terms
      - (i.e., a 1st approximation to a theory of pointed cpo’s)
    - Tensor products
    - Coalesced sums
    - Integer arithmetic with (+, -, *)
    - Linear, real arithmetic (interval arithmetic)
    - Booleans
  - Many properties of (closed) Haskell 98 programs can be formulated in these theories alone
    - Other properties will require additional or more complete theories
      - Induction rules, for instance

# The basic idea for a modular theory solver

- Atomic propositions gleaned from an asserted, closed formula are sorted according to the theories to which they belong
- For each theory, a dedicated solver calculates
  - Conflicts (if any) among the propositions relevant to its theory, or
  - Propositions entailed by the theory, if the solver state is consistent.
- A SAT solver makes tentative truth assignments to the atomic propositions and communicates these to the individual theory solvers
  - The current state is a (partial) assignment to the set of atomic propositions, compatible with truth of the asserted formula
  - A (complete) state that all solvers agree is conflict-free is evidence that the formula is satisfiable
  - If no such state exists, the formula is unsatisfiable
    - A formula $\Phi$ is valid iff the formula $(\neg \Phi)$ is unsatisfiable
    - Modern SAT solvers use sophisticated strategies to quickly prune unsatisfiable search paths

# Example: Normalizing a formula:
## Translation from a closed formula to atomic literals

Formula: | Proxy definitions

forall $x, y.\ x \geq 0 \wedge y \geq 0 \Rightarrow f\,(x + y) \geq 0$

Replace quantified variables by unique constant symbols

$x_0 \geq 0 \wedge y_0 \geq 0 \Rightarrow f\,(x_0 + y_0) \geq 0$

Eliminate implication connective

$\neg\,(x_0 \geq 0) \vee \neg\,(y_0 \geq 0) \vee (f\,(x_0 + y_0) \geq 0)$

Proxy the argument expression in a function application

$\neg\,(x_0 \geq 0) \vee \neg\,(y_0 \geq 0) \vee (f\ v_0 \geq 0)$ — $v_0 = x_0 + y_0$

Proxy the function application in the rightmost inequality

$\neg\,(x_0 \geq 0) \vee \neg\,(y_0 \geq 0) \vee (v_1 \geq 0)$ — $v_0 = x_0 + y_0,\ v_1 = f\ v_0$

Proxy the inequalities

$\neg\, z_0 \vee \neg\, z_1 \vee z_2$ — $v_0 = x_0 + y_0,\ v_1 = f\ v_0,$
$z_0 = x_0 \geq 0,\ z_1 = y_0 \geq 0,$
$z_2 = v_1 \geq 0$

Yielding an equivalent formulation in CNF with all atoms proxied

# Assigning atomic formulas to theory solvers

- Each atomic formula is assigned by a host solver to a particular theory solver for interpretation
  - Operator symbols (which must not be overloaded) are partitioned into sorts corresponding to theories
  - Assignment to a theory follows the sort of the dominant operator symbol of each atomic formula

  Examples:

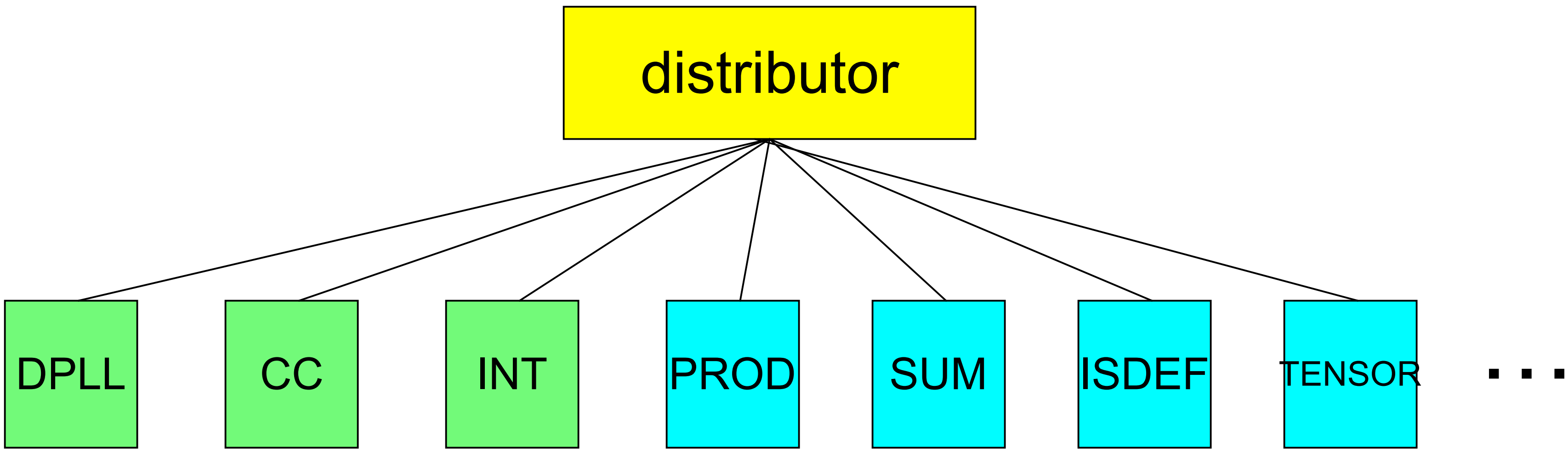
  | | |
  |---|---|
  | $x_0 + y_0$ | : linear arithmetic (INT solver) |
  | $f\ v_0$ | : uninterpreted functions with equality (CC solver) |
  | $x_0 \geq 0$ | : linear arithmetic (INT solver) |
  | … etc. | |

- Theory solvers bind fresh variables as proxies for atomic formulas
  - Each solver reports its set of bound proxy variables to the host solver
    - to establish the data of a working interface

# Modular Architecture of DPT

- *Solver_api* prescribes an *object* template
  - A solver object may have internal state, which is accessed only through its public methods
- A host solver communicates literals of interest to each theory solver
  - An individual theory solver is responsible to detect conflicts among the set of literals it has been given, interpreting only its own theory
  - Detected conflicts are communicated back to the host solver
- A CC (congruence closure) solver propagates equalities
- A SAT solver (DPLL) directs a search for a satisfying assignment to literals extracted from a given formula
  - Backtracks when a conflict is detected in a current assignment
  - Reports satisfiability if a full assignment is made for which no conflict is detected (but doesn't yet trace the satisfying assignment)
  - Reports unsatisfiability if no further assignments are possible and conflict persists

# Architecture of a system of solvers




Modules packaged with DPT

- SAT solver
- Uninterpreted functions w/ equality
- Linear, integer arithmetic
- Real, interval arithmetic

User-defined modules interfaced with DPT …

Cartesian product
Coalesced sum
Strength (approximates definedness)
Tensor product

# Internal architecture of a theory solver

- A typical theory solver has at least three components
  - A *literals* module defines the data representation of literals for this theory solver
    - (a *literal* is either an atomic proposition or its negation)
  - A *core* module implements the decision procedure
    - maintains the state variables of a model for this theory
    - interprets operators of this theory in the model
    - interprets dedicated predicates of this theory (if any)
    - reports conflicts in the state of the model
  - An interface wrapper conforms to the *solver_api*
    - It proxies literals and their subterms with unique variables
      - a proxy map is a bijection between variables and terms
    - Maintains a bijective map between term representations and the equivalent data representations used in an internal model
    - Accepts *set_literal* directives from the host to update the solver state
    - Replies to queries from the host about conflicts detected in the core
    - Manages backtrack requests from the host

# My First Theory Solver: Prod

- First solver: Cartesian product
  - Constants: mkpr :: $t \rightarrow t \rightarrow t$, fst :: $t \rightarrow t$, snd :: $t \rightarrow t$
  - Three axioms can be implemented by reduction rules:
    - fst (mkpr x y) = x
    - snd (mkpr x y) = y
    - (mkpr (fst p) (snd p)) = p
  - Two conditions of inductive definition can be checked
    - (mkpr x y) ≠ x
    - (mkpr x y) ≠ y
  - Prod solver was constructed with a term model
    - Interfaced by following the documented, DPT *solver_api*
      - Reading DPT source code was essential, however
      - Non-critical methods were dummied
    - Given a set of asserted literals, the Prod solver detects any conflict with the axioms and conditions

# A Second Solver: Tensor Product

- The first solver gave me confidence that I knew what I was doing
- So I tried a second solver, for a theory of tensor products in a cpo domain
  - and encountered some surprises!
- The theory is more interesting than Prod
  - Constants: mktr :: t → t → t, tfst :: t → t, tsnd :: t → t
  - Axioms:
    - Isdef y ⇒ tfst (mktr x y) = x
    - Isdef x ⇒ tsnd (mktr x y) = y
    - mktr (tfst p) (tsnd p) = p
  - Inductivity conditions:
    - Isdef x ⇒ x ≠ mktr x y
    - Isdef y ⇒ y ≠ mktr x y
  - where Isdef is an interpreted predicate satisfied by all non-bottom elements of a domain.
- Notice that most of these axioms are implicative formulas

# List of potential conflicts and entailments

- Conflicts:
  - Tr1) Isdef x & x = mktr x y
  - Tr2) Isdef y & y = mktr x y
  - Tr3) Isdef x & x = tfst x
  - Tr4) Isdef y & y = tsnd y
  - Tr5) Isdef z & ⇒ (Isdef (tfst z))
  - Tr6) Isdef z & ⇒ (Isdef (tsnd z))
  - Tr7) Isdef (mktr x y) & ⇒ (Isdef x)
  - Tr8) Isdef (mktr x y) & ⇒ (Isdef y)
  - Tr9) Isdef y & x ≠ tfst (mktr x y)
  - Tr10)Isdef x & y ≠ tsnd (mktr x y)
  - Tr11)Isdef x & Isdef y & ⇒(Isdef (mktr x y))
- Entailments:
  - TI1) x = mktr x y ⇒ ¬ (Isdef x)
  - TI2) y = mktr x y ⇒ ¬ (Isdef y)
  - TI3) x = tfst x ⇒ ¬ (Isdef x)
  - TI4) x = tsnd x ⇒ ¬ (Isdef x)
  - TI5) Isdef z ⇒ Isdef (tfst z)
  - TI6) Isdef z ⇒ Isdef (tsnd z)
  - TI7) Isdef (mktr x y) ⇒ Isdef x
  - TI8) Isdef (mktr x y) ⇒ Isdef y
  - TI9) Isdef y ⇒ x = tfst (mktr x y)
  - TI10)Isdef x ⇒ y = tsnd (mktr x y)
  - TI11) ¬ (Isdef (mktr x y)) ⇒ (¬ (Isdef x) or ¬ (Isdef y))

- All involve the Isdef predicate
- Reduction rules are realized by Tr9, TR10 and TI9 and TI10

# The ubiquitous Isdef suggests managing definedness with a separate theory

- The theory *Strength*
  - Constants:
    - Isdef :: t → prop
  - Axiom:
    - $\neg$ (Isdef x) & $\neg$ (Isdef y) $\Rightarrow$ x = y

- *Strength* is a simple theory for which to build a solver.
  - However, interpreting a proposition (Isdef <term>) can only be done in the particular theory in which <term> is interpreted
  - An Isdef literal must be “shared” between the solver for *Strength* and the solver in which the proposition can be interpreted.
    - Either solver might detect a conflict among asserted literals containing Isdef propositions
      - Similar to equality in this respect
  - The DPT framework provides a mechanism to implement sharing of propositions between individual theory solvers

# Sharing propositions between theory solvers

- Suppose $p$ is a proposition of interest to two theory solvers, $Th_1$ and $Th_2$
- Each solver provides a *proxy* variable for $p$, a name by which it is known to the host framework
  - Suppose $Th_1$ proxies $p$ as $x_1$; $Th_2$ proxies $p$ as $x_2$
  - To indicate to the DPLL solver that the two proxy variables are logically equivalent literals, assert the following clauses to the DPLL solver:
    - ($x_1$ or $\neg\ x_2$) and ($\neg\ x_2$ or $x_1$)
  - That's all there is to it!

# Embedding Strict theories

- There are many useful decision procedures for theories over sets, rather than over a cpo domain
  - In such theories there is no notion of definedness (or not)
  - Examples: linear arithmetic, boolean algebra, etc.
  - When embedded in a pointed cpo domain, the operators of such a theory are said to be *strict* and *total*.
    - Mathematical comment: a subdomain whose algebra consists only of strict operators embeds in a cpo domain as a comonad
- To integrate a decision procedure for a strict theory with a framework for reasoning over cpo's,
  - Require that the variables of each strict operator expression satisfy the Isdef predicate (to assure strictness)
  - Infer that each strict operator expression satisfies Isdef

    (to assure totality)
- This integration can be efficiently implemented in the DPT framework by small additions to the code of the host solver
  - Decision procedures for strict theories remain opaque (abstract)

# What's difficult about this?

- Not much, so long as you stay with decidable theories
  - Comprehensive unit testing is essential
    - it's easy to err on the side of building unnecessary cases into a prototype solver
- What does the future hold?
  - Quantified variable instantiation could be added to DPT
    - There are known algorithms for efficient E-matching (de Moura & Bjorner, 2007), but none has yet been implemented in DPT
  - Traceback reporting
    - The ability to report a satisfying assignment would enable counterexamples to false assertions of validity to be constructed
      - an assignment satisfying ($\neg\ \Phi$) is a counterexample of asserted validity
- To re-implement *Plover*, three more things are needed:
  - a generic theory of induction (and coinduction)
  - an interface to a language front-end, such as *programatica-pfe*
  - termination analysis for recursively-defined functions

# End

# Some references

Sava Krstic and Amit Goel:

**Architecting Solvers for SAT Modulo Theories: Nelson-Oppen with DPLL**

.pdf available from Sava's home page, www.csee.ogi.edu/~krstics/

Grundy, Goel and Krstic: **Decision Procedure Toolkit**

sourceforge.net/projects/dpt

offers downloads of code and documentation;

additional user-submitted documentation is available via the wiki tab

Richard Kieburtz: **P-logic: property verification for Haskell programs**

web.cecs.pdx.edu/~dick/plogic.pdf

Programming logic for a large fragment of Haskell98, with some examples