Modular Type Checking With Decision Procedures

Iavor S. Diatchki

Galois Inc

Question

Can we provide a generic mechanism for integrating decision procedures into the type system of a programming language?

An Observation

- GHC's constraint solver looks a bit like an SMT solver
- SMT: decision procedures cooperate to solve a problem
  - Each algorithm is good at solving one kind of problem
  - Use common coordination logic to:
    - partion original probelm among the decision procedures, and
    - propagate results from one procedure to the rest.

The Theories of Haskell: Equality of Datatypes

- The type checker needs to decide if two types are the same:

```
Maybe a = f Int --> (Maybe = f, a = Int)
Int = Char      --> Impossible
```

- Decidable with first order unification.

The Theories of Haskell: Type Classes

- The type-checker needs to solve class constraints:

```
instance Eq Int
instance Eq a => Eq (Maybe a)

Eq (Maybe Int) --> Eq Int --> ()
```

- Decidable with restrictions on user-defined instances.

The Theories of Haskell: Open Type Families

- The type checker needs to evaluate user-defined type functions:

```
type instance Elem (Maybe a) = a
type instance Elem [a]       = a

Elem (Maybe Int) = x             --> Int = x
(Elem x = Int, Elem x = Char) --> Impossible
```

- Decidable with restrictions on type-family instances.

More Theories of Haskell

- Type-level natural numbers
- Functional dependencies
- Closed type families
- Representational equality
- ... probably more to come ...
  - Injective type functions?
  - Operations on `Symbol`?
  - Type-level integers?

Common language: type variables and ordinary types.

Reasoning in the Combined Theory

- The type-checker is presented with a combined problem:

  ```
  (Eq (Elem (f a)), Maybe a = f (Elem [Int])
  ```

- Here we are using:
  - classes,
  - type families, and
  - type equality.

A modular approach is essential to manage the complexity of the resulting system.

Canonicalization: Partitioning the Problem

- Transform the problem so that each constraint belongs to a single theory
- This is done by naming terms that belong to a foreign theory:

```
(Eq (Elem (f a)), Maybe a = f (Elem [Int])
-->
( Eq x                    -- type classes
, Elem (f a) = x          -- type families
, Maybe a = f y           -- type equality
, Elem [Int] = y          -- type families
)
```

A Solver's Responses

- For each new constraint a solver may:
  - report an inconsistency (i.e., we found an error);
  - discharge the constraint, maybe adding extra sub-goals;
  - give up, storing the constraint for later use.
- Example

```
( Eq x               -- give up
, Elem (f a) = x     -- give up
, Maybe a = f y      --> solve if (Maybe = f, a = y)
, Elem [Int] = y     --> solve if Int = y
)
```

Communication Between Solvers

- Of particular interest are subgoals of the form `x = t`
- `t` is a "simple" type, understood by all theories.
- These may be used to rewrite existing constraints, which may enable further progress

```
( Eq x, Elem (Maybe Int) = x, f = Maybe, y = a = Int)
( Eq Int, x = Int,            f = Maybe, y = a = Int)
(         x = Int,            f = Maybe, y = a = Int)
```

Solver For Natural Numbers

- Assume an existing decision procedure:

```
sat (x + 2 = y) == Sat { x = 0, y = 2 }
sat (2 + 3 = 6) == Unsat
```

- Satisfying assignment contains concrete numbers
- A useful wrapper:

```
prove p = sat (Not p) == Unsat
```

- Example:

```
prove (2 + 3 = 5) == True
prove (x + y = z) == False
```

Adding a New Constraint

We want to add a new constraint `P`, to an existing set of stuck constraints `Ps` (the *inert* set in GHC lingo).

1. Check for redundancy

```
if prove (Ps => P) then Done
                   else ...
```

2. Check for consistency

```
case sat (Ps && P) of
  Unsat -> report error
  Sat su -> ...
```

Improvement

- Adding `P` to the inert set may result in opportunities for improvement
- If `prove (Ps && P => x = t)`
  - `t` is simple, and `x` in `fvs(Ps && P)`
- then we add a new sub-goal `x=t`
  - The new goal does not make the problem harder
  - It ensures progress: a variable gets instantiated
- How to find `t`?

Improvement With Concrete Values

3. Check for improvements with ground type. We have candidates from the consistency check:

```
[  x = n | (x,n) <- su, prove (Ps && P => x = n) ]
```

- Example:

```
Ps = (), P = (5 + x = 8)

sat (5 + x = 8) == Sat { x = 3 } &&
prove (5 + x = 8 => x = 3)

new sub-goal: x = 3
```

Improvement With Variables

4. For any distinct `x` and `y` in `fvs(Ps && P)`:

   ```
   [ x = y | prove (Ps && P => x = y) ]
   ```

- For example, consider a constraint like `x + 0 = y`:
  - Improvement with ground values fails (no unique solution)
  - However, `prove (x + 0 = y => x = y)` is `True`
  - new sub-goal: $(x = y)$

Finally: Simplify Delayed Constraints

5. Check if we can discharge existing delayed constraints, using the new constraint: `check P Ps`

```
check done (q:qs)

  | prove (done && qs => q) = do discharge q
                                 check done qs

  | otherwise               = check (q && done) qs

check done [] = return done
```

Conclusion

- A practical implementations should probably optimize things:
  - Avoid calls to decision procedure for common simple cases (e.g., evaluation)
  - Lazy canonicalization
  - Combine multiple solver steps into a single step.
- The technique did not make essential use of the fact that we are working with numbers
- It'd be interesting to provide a general mechanism for integrating decision procedures in a language's type-checker.

Given and Wanted Constraints

- Given constraints do not need to be discharged
  - they state known facts
- They arise from type signatures, existentials, GADTs
- Processed in a similar way:
  - Inconsistency indicates unreachable code
  - Improvement with other givens results in new givens
  - No need to keep them minimal, so we can skip step 5
  - Adding a given kciks-out all wanteds

Evidence

- Usually decision procedures do not produce proofs
- Proofs could be large, often they involve search
- One option:
    - "oracle" proofs, just record call to decision procedure
    - only store facts that the decision procedure used?