Untyped Types

May 11, 2009

Rockwell Collins

Advanced Technology Center

Goals

- Compositional
  - Reasoning approach for Complex and Simple Types is the Same
  - Complex types can be built from simple types
    - And Disabled
- Uniform
  - Amenable to Automation
- Efficient
  - Minimize Time (Search)
  - Minimize Space (Size)
- Complete
  - If problem is decidable, solution should work

Tension!

```
(defun Type (x)
  (and (TypeA x) (TypeB x)))

(defun OType (x)
  (or (Type x) (integerp x))

(defthm Type-fn
  (Type (fn z))

=> (TypeA (fn z)) ??
=> (OType (fn z)) ??
```

Type Reasoning in ACL2

A special pass

To relieve a hypothesis we only use type reasoning, evaluation of ground terms, and presence among our known assumptions, no rewriting (including no opening of definitions)

Beware of non-recursive functions occurring in the hypotheses of :type-prescription rules!

If it is enabled, you are screwed
If it is disabled, you are screwed

Can we avoid being screwed ?

Principles

- :forward-chaining is the workhorse
  - Minimizes search (efficient?)
  - Adds predicates to type-alist
    - No new structure in :forward-chaining rules (size)
    - type-alist size should be bounded
- Use :type-prescription only in desperation
  - Fights against :forward-chaining (inefficient)
  - Experimentally slow
- Use :rewrite (only) when there is no search required
  - Ideally :rewrite is not needed

Conjunction (And) Type

```
(defthm Type-implies
  (implies
    (Type x)
    (and (TypeA x)
         (TypeB x)))
  :rule-classes (:forward-chaining))
```

Expensive :rewrite rule because **many** types could imply (TypeA x)

```
(defthm implies-Type
  (implies
    (and
      (TypeA x)
      (TypeB x))
    (Type x))
  :rule-classes (:rewrite (:forward-chaining
                           :trigger-terms
                           ((TypeA x)
                            (TypeB x)))))
```

Ideally these :rewrites will never be used

```
(defthm not-Type-implies
  (and
    (implies
      (and (not (Type x))
           (TypeA x))
      (not (TypeB x)))
    (implies
      (and (not (Type x))
           (TypeB x))
      (not (TypeA x))))
  :rule-classes (:forward-chaining))
```

```
(defthm implies-not-Type
  (and (implies
         (not (TypeA x))
         (not (Type x)))
       (implies
         (not (TypeB x))
         (not (Type x))))
  :rule-classes (:rewrite
                 :forward-chaining))
```

Disjunction (Or) Type

```
(defthm not-Type-implies
  (implies
    (not (Type x))
    (and (not (TypeA x))
         (not (TypeB x))))
  :rule-classes (:forward-chaining))

(defthm implies-not-Type
  (implies
    (and
      (not (TypeA x))
      (not (TypeB x)))
    (not (Type x)))
  :rule-classes (:rewrite (:forward-chaining
                           :trigger-terms
                           ((TypeA x)
                            (TypeB x)))))
```

Negated Types .. Established by :type-prescription + :forward-chaining

```
(defthm Type-implies
  (and
    (implies
      (and (Type x)
           (not (TypeA x)))
      (TypeB x))
    (implies
      (and (Type x)
           (not (TypeB x)))
      (TypeA x)))
  :rule-classes (:forward-chaining))

(defthm implies-Type
  (and (implies
         (TypeA x)
         (Type x)
         (implies
           (TypeB x)
           (Type x)))
  :rule-classes (:rewrite
                 :forward-chaining))
```

Nominal Data Structure Types

```
(defthm implies-str-p
  (implies
    (and
      (tag-equal x 'str)
      (true-size x 3)
      (typeA (field-A x))
      (typeB (field-B x)))
    (str-p x))
  :rule-classes (:rewrite
  (:forward-chaining :trigger-terms
  ((field-A x) (field-B x)))))
```

It is convenient to have tagged structures

```
(defthm str-p-implies-car-equal
  (implies
    (str-p x)
    (and (tag-equal x 'str)
         (true-size x 3)))
  :rule-classes (:forward-chaining))
```

```
(defthm str-p-implies
  (implies
    (str-p x)
    (and (typeA (field-A x))
         (typeB (field-B x))))
  :rule-classes (:rewrite
  (:forward-chaining :trigger-terms
    ((field-A x) (field-B x)))))
```

```
(defthm str-p-str
  (implies
    (and (typeA A)
         (typeB B))
    (str-p (str A B)))
  :rule-classes (:rewrite
 (:forward-chaining :trigger-terms
  ((str A B)))))
```

(Negated) Nominal Type

```
(defthm not-tag-implies-not-str-p
  (implies
    (and
      (tag-equal x tag)
      (not (equal tag 'str)))
    (not (str-p x)))
:rule-classes (:type-prescription))
```

Note that tag is free

Presumably added by some other nominal type ..

Evaluation of a constant term

:rewrite is not used during type reasoning

:forward-chaining would add the negation of every known nominal type (potentially very large)

:type-prescription is our only option

Function Signatures

```
(defthm fn-signature
  (implies
    (and
      (TypeX a)
      (TypeY b))
    (Type (fn a b)))
  :rule-classes (:rewrite
  (:forward-chaining :trigger-terms ((fn a b)))))
```

```
(defun Type (x)
  (and (TypeA x)
       (TypeB x)))

(defun OType (x)
  (or (Type x) ..))
```

```
(Type (fn a b))
(TypeX a)
(TypeY b)
|-
```

Implies-OType

```
(OType (fn a b))
(Type (fn a b))
(TypeA (fn a b))
(TypeB (fn a b))
(TypeX a)
(TypeY b)
|-
```

Type-implies

Backchaining Backbreaker

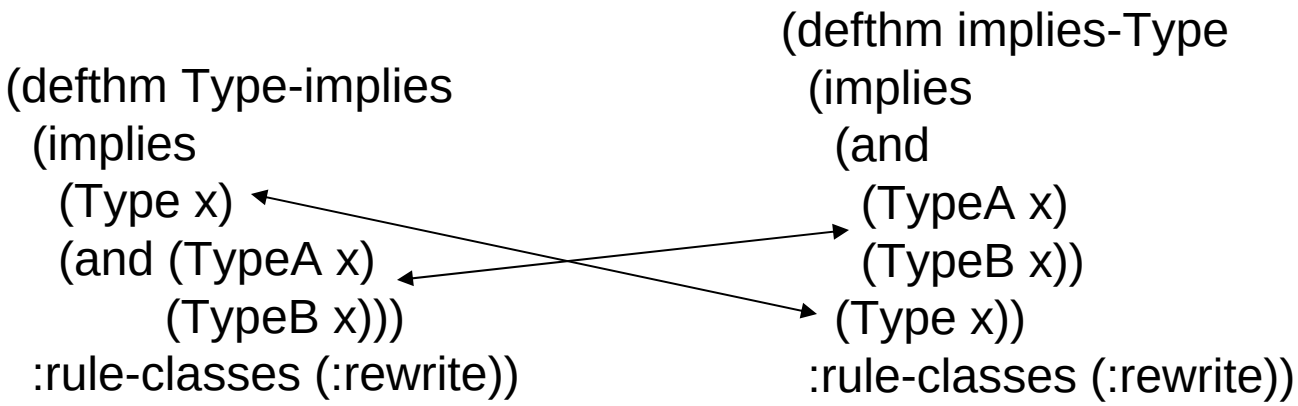
- When ACL2 asks (Type x) during back chaining
  - X is a constant
    - Hopefully type is executable
    - Otherwise x is treated as an expression
  - X is a symbol
    - Appears in type-alist
    - Enough information in the type-alist to deduce by type reasoning
  - X is an expression (function application)
    - Appears in type-alist
    - Introduced new structure in hypothesis (or ancestor RHS)
      - :forward-chaining does not apply during back chaining
      - Requires a :rewrite rule to trigger on (Type (fn ..))
      - (OType (fn ..)) ? (TypeA (fn ..)) ? (screwed again)

        » Only resolution is to employ :rewrite rules that do search

        » Make Type-implies and not-Type-implies :rewrites

Heuristically Challenged

- Heuristics (ancestors check) will save us from circular rewrites ..



- But they bite us during :forward-chaining
  - We promised: no new structure when :forward-chaining
    - ACL2 doesn't believe us
    - Heuristics can defeat :forward-chaining rules under certain conditions

Principles (Revised)

- :forward-chaining is the workhorse
  - Minimizes search (efficient?)
  - Adds predicates to type-alist
    - No new structure in :forward-chaining rules (size)
    - type-alist size should be bounded
- Use :type-prescription only in desperation
  - Fights against :forward-chaining (inefficient)
  - Experimentally slow
- Use :rewrite in addition to :forward-chaining
  - To address backchaining issues

Questions

- How do we estimate the cost of a :forward-chaining rule?
- What is the performance impact of each new type-alist entry?
- Have we made good time/space tradeoffs?
- Can we do better?

Type-alist Fixedpoints and Structure

```
(defthm subset-append
  (and
    (setp x)
    (setp y))
  (and (subset x (append x y))
       (subset y (append x y)))
  :rule-classes :forward-chaining)
```

Introduces new structure

No fixedpoint

```
(defthm subset-append
  (and
    (setp x)
    (setp y))
  (and (subset x (append x y))
       (subset y (append x y)))
  :rule-classes (:forward-chaining
                 :trigger-terms ((append x y))))
```

Structure already exists

Note: More efficient to use domain specific (union x y)