CS576 Topics in Automated Deduction

Elsa L Gunter
2112 SC, UIUC
egunter@illinois.edu
http://courses.engr.illinois.edu/cs576

Slides based in part on slides by Tobias Nipkow

March 5, 2015

Format for Inductive Relations Definitions

inductive R :: "$\tau \longrightarrow$ bool" where

$[\![R(a_{1,1}); \ldots; R(a_{1,n}); A_{1,1}; \ldots; A_{1,k}]\!] \Longrightarrow R(a_1)$ |

... |

$[\![R(a_{m,1}); \ldots; R(a_{m,1}); A_{m,1}; \ldots; A_{m,j}]\!] \Longrightarrow R(a_m)$

where $A_{i,j}$ are side conditions not involving R.

Format for Inductive Relations Definitions

inductive R :: "$\tau \longrightarrow$ bool" where
  $[\![R(a_{1,1}); \ldots; R(a_{1,n}); A_{1,1}; \ldots; A_{1,k}]\!] \Longrightarrow R(a_1)$ |
    ... |
  $[\![R(a_{m,1}); \ldots; R(a_{m,1}); A_{m,1}; \ldots; A_{m,j}]\!] \Longrightarrow R(a_m)$

where $A_{i,j}$ are side conditions not involving R.

Format for Mutual Inductive Relations Definitions

inductive
R_1 :: "$\tau_1 \longrightarrow$ bool" and
...
R_n :: "$\tau_n \longrightarrow$ bool" where
  $[\![R_i(a_{1,1}); \ldots; R_j(a_{1,n}); A_{1,1}; \ldots; A_{1,k}]\!] \Longrightarrow R_k(a_1)$ |
    ... |
  $[\![R_m(a_{m,1}); \ldots; R_n(a_{m,1}); A_{m,1}; \ldots; A_{m,j}]\!] \Longrightarrow R_p(a_m)$

where $A_{i,j}$ are side conditions not involving any R_k.

Example with Mutual Recursion

```
inductive
Even ::  "nat ⇒ bool" and
Odd ::  "nat ⇒ bool" where
ZeroEven [intro!]:  "Even 0" |
OddOne [intro!]:  "Odd (Suc 0)" |
OddSucEven [intro]:  "Odd n ⟹ Even (Suc n)" |
EvenSucOdd [intro]:  "Even n ⟹ Odd (Suc n)"
```


General Recursive Functions: fun

Example:

fun fib :: "nat ⇒ nat" where
  "fib 0 = 0" |
  "fib 1 = 1" |
  "fib (Suc(Suc x)) = (fib x + fib (Suc x))"

Not primitive recursive because of `fib(Suc(Suc x))` on left, and because of `fib(Suc x)` on right.

`fun`: Rules of Use

Compared to `primrec`, very few restrictions:

- Can be used to define functions over any type
- Clauses in `fun` must be equations
- Left-hand side is function being defined applied to terms built from data constructors, distinct variables and wildcards
- Right-hand side is a expression made from the function being defined, the variables in the argument on the left, and previously defined terms
- If clauses overlap, first takes precedence.
- Calculates a measure from lexicographic ordering of some collection of arguments

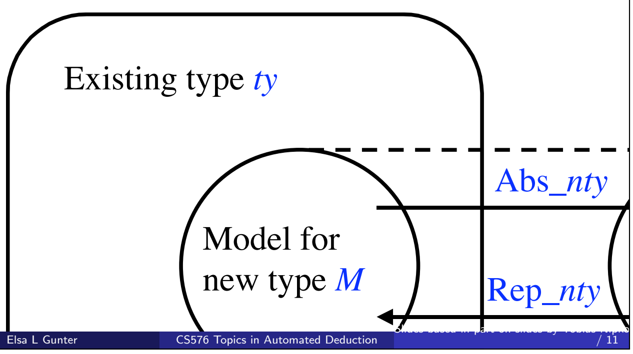
Example: `sep`

Define a function for putting a separator between all adjacent elements in a list:

```
fun sep ::  "'a * 'a list => 'a list" where
 "sep(a, []) = []" |
 "sep(a, [x]) = [x]" |
 "sep(a, x#y#zs) = x # a # sep(a,y#zs)"
```


Demo: General Recursive Functions

Introducing new types



Properties of a Defined Type Syntax for `typedef`:

$$\texttt{typedef}\ nty = \text{"}modeling\_set\text{"}$$
$$\textit{Proof of } \exists x. x \in modeling\_set.$$

introduces a new type named *nty*, and functions

$$\texttt{Abs\_}nty :: ty \Rightarrow new\_ty$$
$$\texttt{Rep\_}nty :: new\_ty \Rightarrow ty$$

where *modeling_set*::*ty*

Properties of a Defined Type Theorems automatically provided:

Rep_*nty*: $\texttt{Rep\_}nty\ x \in modeling\_set$
Abs_*nty*_inverse: $\texttt{Abs\_}nty(\texttt{Rep\_}nty\ x) = x$
Rep_*nty*_inverse:
$y \in modeling\_set \Longrightarrow \texttt{Rep\_}nty(\texttt{Abs\_}nty\ y) = y$
Abs_*nty*_inject:
$[|x \in modeling\_set : y \in modeling\_set|] \Longrightarrow$
$(\texttt{Abs\_}nty\ x = \texttt{Abs\_}nty\ y) = (x = y)$
Rep_*nty*_inject: $(\texttt{Rep\_}nty\ x = \texttt{Rep\_}nty\ y) = (x = y)$