# Computer Supported Modeling and Reasoning

David Basin, Achim D. Brucker, Jan-Georg Smaus, and Burkhart Wolff

April 2005

http://www.infsec.ethz.ch/education/permanent/csmr/

# Higher-Order Logic: Conservative Extensions

Burkhart Wolff

# Outline

In the previous lecture, we have derived all well-known inference rules. There is now the need to scale up. Today we look at conservative theory extensions, an important method for this purpose.

In the weeks to come, we will look at how mathematics is encoded in the Isabelle/HOL library.

# Conservative Theory Extensions: Basics

Basic definitions (c.f. [GM93]):

**Definition 1 (theory):**

A (syntactic) theory $T$ is a triple $(\chi, \Sigma, A)$, where $\chi$ is a type signature, $\Sigma$ a signature and $A$ and a set of axioms.

**Definition 2 (theory extension):**

A theory $T' = (\chi', \Sigma', A')$ is an extension of a theory $T = (\chi, \Sigma, A)$ iff $\chi \subseteq \chi'$ and $\Sigma \subseteq \Sigma'$ and $A \subseteq A'$.

# Definitions (Cont.)

**Definition 3 (conservative extension):**

A theory extension $T' = (\chi', \Sigma', A')$ of a theory $T = (\chi, \Sigma, A)$ is conservative iff for the set of provable formulas $Th$ we have

$$Th(T) = Th(T') \mid_{\Sigma},$$

where $\mid_{\Sigma}$ filters away all formulas not belonging to $\Sigma$.

Counterexample:

$$\overline{\forall f :: \alpha \Rightarrow \alpha.\, Y\; f = f\; (Y\; f)}\ \text{fix}$$

# Consistency Preserved

**Corollary 1 (consistency):**

If $T'$ is a conservative extension of $T$, then

$$False \notin Th(T) \Rightarrow False \notin Th(T').$$

# Syntactic Schemata for Conservative Extensions

- Constant definition
- Type definition
- Constant specification
- Type specification

Will look at first two schemata now.

For the other two see [GM93].

# Constant Definition

**Definition 4 (constant definition):**

A theory extension $T' = (\chi', \Sigma', A')$ of a theory $T = (\chi, \Sigma, A)$ is a constant definition, iff

- $\chi' = \chi$ and $\Sigma' = \Sigma \cup \{c :: \tau\}$, where $c \notin dom(\Sigma)$;
- $A' = A \cup \{c = E\}$;
- $E$ does not contain $c$ and is closed;
- no subterm of $E$ has a type containing a type variable that is not contained in the type of $c$.

# Constant Definitions Are Conservative

**Lemma 1 (constant definitions):**

Constant definitions are conservative.

Proof Sketch:

- $Th(T) \subseteq Th(T') \mid_\Sigma$ : trivial.
- $Th(T) \supseteq Th(T') \mid_\Sigma$ : let $\pi'$ be a proof for $\phi \in Th(T') \mid_\Sigma$. We unfold any subterm in $\pi'$ that contains $c$ via $c = E$ into $\pi$. Then $\pi$ must be a proof in $T$, implying $\phi \in Th(T)$.

# Side Conditions

Where are those side conditions needed? What goes wrong?

Very simple example: Let $E \equiv \exists x :: \alpha y :: \alpha.\, x \neq y$ and suppose $\sigma$ is a type inhabited by only one term, and $\tau$ is a type inhabited by at least two terms. Then we would have:

$$
\begin{array}{ll}
 & c = c \qquad \text{holds by } \textit{refl} \\
\Longrightarrow & (\exists x :: \sigma\, y :: \sigma.\, x \neq y) = (\exists x :: \tau\, y :: \tau.\, x \neq y) \\
\Longrightarrow & False = True \\
\Longrightarrow & False
\end{array}
$$

This explains definition of $True$. Other (standard) example later.

# Constant Definition: Examples

Definitions of $True$, $False$, $\wedge$, $\vee$, $\forall$ revisited.

| | | |
|---|---|---|
| True_def: | True | $\equiv ((\lambda x{::}bool.\ x) = (\lambda x.\ x))$ |
| All_def : | All (P) | $\equiv (P = (\lambda x.\ True))$ |
| Ex_def: | Ex(P) | $\equiv \forall Q.\ (\forall x.\ P\ x \longrightarrow Q) \longrightarrow Q$ |
| False_def : | False | $\equiv (\forall P.\ P)$ |
| not_def: | $\neg$ P | $\equiv P \longrightarrow False$ |
| and_def: | P $\wedge$ Q | $\equiv \forall R.\ (P \longrightarrow Q \longrightarrow R) \longrightarrow R$ |
| or_def : | P $\vee$ Q | $\equiv \forall R.\ (P \longrightarrow R) \longrightarrow (Q \longrightarrow R) \longrightarrow R$ |
| if_def : | If P x y | $\equiv THE\ z{::}'a.\ (P{=}True \longrightarrow z{=}x) \wedge (P{=}False \longrightarrow z{=}y)$ |

Recall that All (P) is syntactically equivalent to $\forall$ x. P x,
Ex(P) equivalent to $\exists$ x. P x.

# More Constant Definitions in Isabelle

**let**−**in**−, if−then−else, unique existence:

**consts**

```
Let  ::  ['a, 'a ⇒ 'b] ⇒ 'b
If   ::  [bool, 'a, 'a] ⇒ 'a
```

**defs**

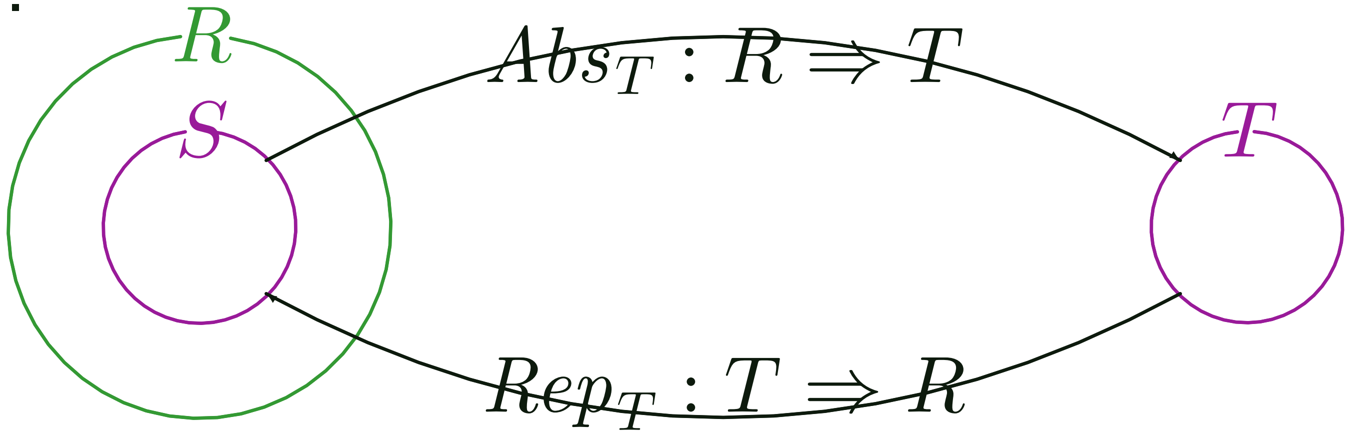
```
Let_def "Let s f ≡ f(s)"
if_def   "If P x y ≡ THE z::'a.(P=True⇒z=x) ∧
                                    (P=False⇒z=y)"
Ex1_def "Ex1(P) ≡∃x. P(x) ∧ (∀y. P(y) ⇒ y=x)"
```

Recall: ⇒ is function type arrow; also recall syntax for [...] ⇒ ... .

# Type Definitions

Type definitions, explained intuitively: we have

- an existing type $R$;
- a predicate $S : R \Rightarrow bool$, defining a non-empty "subset" of $R$;
- axioms stating an isomorphism between $S$ and the new type $T$.



# Type Definition: Definition

**Definition 5 (type definition):**

Assume a theory $Th = (\chi, \Sigma, A)$ and a type $R$ and a term $S$ such that $\Sigma \vdash S : R \Rightarrow bool$.

A theory extension $Th' = (\chi', \Sigma', A')$ of $Th$ is a type definition for type $T$ (where $T$ fresh), iff

$$\begin{array}{rcll} \chi' & = & \chi & \uplus \; \{T\}, \\ \Sigma' & = & \Sigma & \cup \; \{Abs_T : R \Rightarrow T, Rep_T : T \Rightarrow R\} \\ A' & = & A & \cup \; \{\forall x. Abs_T(Rep_T\, x) = x, \\ & & & \quad \forall x. S\, x \Rightarrow Rep_T(Abs_T\, x) = x\} \end{array}$$

Proof obligation $\exists x.\, S\, x$ can be proven inside HOL!

# Type Definitions Are Conservative

**Lemma 2 (type definitions):**

Type definitions are conservative.

Proof see [GM93, pp.230].

# HOL Is Rich Enough!

This may seem fishy: if a new type is always isomorphic to a subset of an existing type, how is this construction going to lead to a "rich" collection of types for large-scale applications?

But in fact, due to $ind$ and $\Rightarrow$, the types in HOL are already very rich.

We now give three examples revealing the power of type definitions.

# Example: Typed Sets

General scheme, substituting $R \equiv \alpha \Rightarrow bool$ ($\alpha$ is any type variable), $T \equiv \alpha\ set$ (or $set$), $S \equiv \lambda x :: \alpha \Rightarrow bool.\, True$

$$\begin{array}{rcll}
\chi' & = & \chi & \uplus \{set\}, \\
\Sigma' & = & \Sigma & \cup \{Abs_{set} : (\alpha \Rightarrow bool) \Rightarrow \alpha\ set, \\
 & & & \quad Rep_{set} : \alpha\ set \Rightarrow (\alpha \Rightarrow bool)\} \\
A' & = & A & \cup \{\forall x. Abs_{set}(Rep_{set}\, x) = x, \\
 & & & \quad \forall x. True \Rightarrow Rep_{set}(Abs_{set}\, x) = x\}
\end{array}$$

Simplification since $S \equiv \lambda x.\, True$. Proof obligation: $(\exists x.\, S\, x)$ trivial since $(\exists x.\, True) = True$. Inhabitation is crucial!

# Sets: Remarks

Any function $r : \tau \Rightarrow bool$ can be interpreted as a set of $\tau$; $r$ is called characteristic function. That's what $Abs_{set}\ r$ does; $Abs_{set}$ is a wrapper saying "interpret $r$ as set".

$S \equiv \lambda x.True$ and so $S$ is trivial in this case.

# More Constants for Sets

For convenient use of sets, we define more constants:

$$\begin{array}{rcl} \{x \mid f\,x\} & \cong & Collect\ f = Abs_{set}\ f \\ x \in A & = & (Rep_{set}\ A)\ x \\ A \cup B & = & \{x \mid x \in A \vee x \in B\} \\ & \vdots & \end{array}$$

Consistent set theory adequate for most of mathematics and computer science !

Here, sets are just an example to demonstrate type definitions. Later we study them for their own sake.

# Example: Pairs

Consider type $\alpha \Rightarrow \beta \Rightarrow bool$. We can regard a term $f : \alpha \Rightarrow \beta \Rightarrow bool$ as a representation of the pair $(a, b)$, where $a :: \alpha$ and $b :: \beta$, iff $f\, x\, y$ is true exactly for $x = a$ and $y = b$. Observe:

- For given $a$ and $b$, there is exactly one such $f$ (namely, $\lambda x :: \alpha\, y :: \beta.x = a \wedge y = b$).
- Some functions of type $\alpha \Rightarrow \beta \Rightarrow bool$ represent pairs and others don't (e.g., the function $\lambda xy.True$ does not represent a pair). The ones that do are exactly the ones that have the form $\lambda x :: \alpha\, y :: \beta.\, x = a \wedge y = b$, for some $a$ and $b$.

# Type Definition for Pairs

This gives rise to a type definition where $S$ is non-trivial:

$$
\begin{aligned}
R &\equiv \alpha \Rightarrow \beta \Rightarrow bool \\
S &\equiv \lambda f :: \alpha \Rightarrow \beta \Rightarrow bool. \\
&\qquad \exists ab.\, f = \lambda x :: \alpha\, y :: \beta . x = a \wedge y = b \\
T &\equiv \alpha \times \beta \qquad\qquad (\times \text{ infix})
\end{aligned}
$$

It is convenient to define a constant Pair_Rep (not to be confused with $Rep_\times$) as follows: Then
Pair_Rep a b = $\lambda$x ::' a y ::' b. x=a $\wedge$y=b.

# Implementation in Isabelle

Isabelle provides a special syntax for type definitions:

**typedef** (T)
    (typevars) T' = "{x. A(x)}"

How is this linked to our scheme:

- the new type is called $T'$;
- $R$ is the type of $x$ (inferred);
- $S$ is $\lambda x.\, A\, x$;
- constants Abs_$T$ and Rep_$T$ are automatically generated.

# Isabelle Syntax for Pair Example

**constdefs**

Pair_Rep :: ['a, 'b] ⇒ ['a, 'b] ⇒ bool
"Pair_Rep ≡ (λ a b. λ x y. x=a ∧y=b)"

**typedef** (Prod)

('a, 'b) "∗" ( infixr 20) =
"{f.∃ a b. f=Pair_Rep(a::'a)(b ::' b)}"

The keyword `constdefs` introduces a constant definition. The definition and use of Pair_Rep is for convenience. There are "two names" ∗ and Prod.

See Product_Type.thy.

# Example: Sums

An element of $(\alpha, \beta)$ `sum` is either Inl a ::' a or Inr b ::' b.

Consider type $\alpha \Rightarrow \beta \Rightarrow bool \Rightarrow bool$. We can regard $f : \alpha \Rightarrow \beta \Rightarrow bool \Rightarrow bool$ as a

| representation of ... | iff $f\,x\,y\,i$ is true for ... |
|---|---|
| $Inl\ a$ | $x = a$, $y$ arbitrary, and $i = True$ |
| $Inr\ b$ | $x$ arbitrary, $y = b$, and $i = False$. |

Similar to pairs.

# Isabelle Syntax for Sum Example

**constdefs**

```
  Inl_Rep :: ['a, 'a, 'b, bool] ⇒ bool
 "Inl_Rep ≡ (λa. λx y p. x=a ∧ p)"
  Inr_Rep :: ['b, 'a, 'b, bool] ⇒ bool
 "Inr_Rep ≡ (λb. λx y p. y=b ∧ ¬p)"
typedef (Sum)
 ('a,'b)"+" =
   "{f. (∃ a. f = Inl_Rep(a::'a)) ∨
        (∃ b. f = Inr_Rep(b::'b))}"
```

See Sum_Type.thy.

Exercise: How would you define a type `even` based on `nat`?

# Summary

- We have introduced a method to safely build up larger theories
- . . . and built sums and products
- . . . and sets !
  (i.e. we have a method to overcome
  the problem of inconsistencies for
  the crucial problems !)

# More Detailed Explanations

# Axioms or Rules

Inside Isabelle, axioms are `thm`'s, and they may include Isabelle's metalevel implication $\Longrightarrow$. For this reason, it is not required to mention rules explicitly.

But speaking more generally about HOL, not just its Isabelle implementation, one should better say "rules" here, i.e., objects with a horizontal line and zero or more formulas above the line and one formula below the line.

# Provable Formulas

The provable formulas are terms of type $bool$ derivable using the inference rules of HOL and the empty assumption list. We write $Th(T)$ for the derivable formulas of a theory $T$.

# Closed Terms

A term is closed or ground if it does not contain any free variables.

# Definition of $True$ Is Type-Closed

$True$ is defined as $\lambda x :: bool.\, x = \lambda x.\, x$ and not $\lambda x :: \alpha.x = \lambda x.\, x$. The definition must be type-closed.

# Fixpoint Combinator

Given a function $f : \alpha \Rightarrow \alpha$, a fixpoint of $f$ is a term $t$ such that $f\,t = t$. Now $Y$ is supposed to be a fixpoint combinator, i.e., for any function $f$, the term $Y\,f$ should be a fixpoint of $f$. This is what the rule

$$\frac{}{\forall f :: \alpha \Rightarrow \alpha . Y\,f = f\,(Y\,f)}\;\text{fix}$$

says. Consider the example $f \equiv \neg$. Then the axiom allows us to infer $Y(\neg) = \neg(Y(\neg))$, and it is easy to derive $False$ from this. This axiom is a standard example of a non-conservative extension of a theory.

This inconsistency is not surprising: Not every function has a fixpoint, so there cannot be a combinator returning a fixpoint of any function.

Nevertheless, fixpoints are important and must be realized in some way, as we will see later.

# Side Conditions

By side conditions we mean

- $E$ does not contain $c$ and is closed;
- no subterm of $E$ has a type containing a type variable that is not contained in the type of $c$;

in the definition.

The second condition also has a name: one says that the definition must be type-closed.

The notion of having a type is defined by the type assignment calculus. Since $E$ is required to be closed, all variables occurring in $E$ must be $\lambda$-bound, and so the type of those variables is given by the type superscripts.

# Domains of $\Sigma$, $\Gamma$

The domain of $\Sigma$, denoted $dom(\Sigma)$, is $\{c \mid (c :: A) \in \Sigma \text{ for some } A\}$.

Likewise, the domain of $\Gamma$, denoted $dom(\Gamma)$, is $\{x \mid (x :: A) \in \Gamma \text{ for some } A\}$.

Note the slight abuse of notation.

# constdefs

In Isabelle theory files, `consts` is the keyword preceding a sequence of constant declarations (i.e., this is where the $\Sigma$ is defined), and `defs` is the keyword preceding the constant definitions defining these constants (i.e., this is where the $A$ is defined.

`constdefs` combines the two, i.e. it allows for a sequence of both constant declarations and definitions, and the theorem identifier $c\_def$ is generated automatically. E.g.

**constdefs**

id :: "'a ⇒ 'a"

"id ≡ λ x. x"

will bind `id_def` to $id \equiv \lambda x.x$.

# $S$

Here, $S$ is any “predicate”, i.e., term of type $R \Rightarrow bool$, not necessarily a constant.

# Fresh $T$

The type constructor $T$ must not occur in $\chi$.

# What Is $T$?

We use the letter $\chi$ to denote the set of type constructors (where the arity and fixity is indicated in some way). So since $T \in \chi'$, we have that $T$ should be a type constructor. However, we abuse notation and also use $T$ for the type obtained by applying the type constructor $T$ to a vector of different type variables (as many as $T$ requires).

## $\uplus$

The symbol $\uplus$ denotes disjoint union, so the expression $A \uplus B$ is well-formed only when $A$ and $B$ have no elements in common.

# What Are $Abs_T$ and $Rep_T$?

Of course we are giving a schematic definition here, so any letters we use are meta-notation.

Notice that $Abs_T$ and $Rep_T$ stand for new constants. For any new type $T$ to be defined, two such constants must be added to the signature to provide a generic way of obtaining terms of the new type. Since the new type is isomorphic to the "subset" $S$, whose members are of type $R$, one can say that $Abs_T$ and $Rep_T$ provide a type conversion between (the subset $S$ of) $R$ and $T$.

So we have a new type $T$, and we can obtain members of the new type by applying $Abs_T$ to a term $t$ of type $R$ for which $S\,t$ holds.

# Isomorphism

The formulas

$$\forall x. Abs_T(Rep_T\, x) = x$$
$$\forall x. S\, x \Rightarrow Rep_T(Abs_T\, x) = x$$

state that the "set" $S$ and the new type $T$ are isomorphic. Note that $Abs_T$ should not be applied to a term not in "set" $S$. Therefore we have the premise $S\, x$ in the above equation.

Note also that $S$ could be the "trivial filter" $\lambda x. True$. In this case, $Abs_T$ and $Rep_T$ would provide an isomorphism between the entire type $R$ and the new type $T$.

# Proof Obligation

We have said previously that $S$ should be a non-empty "subset" of $T$. Therefore it must be proven that $\exists x.\, S\, x$. This is related to the semantics.

Whenever a type definition is introduced in Isabelle, the proof obligation must be shown inside Isabelle/HOL. Isabelle provides the `typedef` syntax for type definitions, as we will see later.

# Inhabitation in the $set$ Example

We have $S \equiv \lambda x :: \alpha \Rightarrow bool.\, True$, and so in $(\exists x. Sx)$, the variable $x$ has type $\alpha \Rightarrow bool$. The proposition $(\exists x. Sx)$ is true since the type $\alpha \Rightarrow bool$ is inhabited, e.g. by the term $\lambda x :: \alpha.\, True$ or $\lambda x :: \alpha.\, False$.

Beware of a confusion: This does not mean that the new type $\alpha\, set$, defined by this construction, is the type of non-empty sets. There is a term for the empty set: The empty set is the term $Abs_{set}\ (\lambda x.\, False)$.

Recall a previous argument for the importance of inhabitation.

# Trivial $S$

We said that in the general formalism for defining a new type, there is a term $S$ of type $R \Rightarrow bool$ that defines a "subset" of a type $R$. In other words, it filters some terms from type $R$. Thus the idea that a predicate can be interpreted as a set is present in the general formalism for defining a new type.

Now we are talking about a particular example, the type $\alpha\, set$. Having the idea "predicates are sets" in mind, one is tempted to think that in the particular example, $S$ will take the role of defining particular sets, i.e., terms of type $\alpha\, set$. This is not the case!

Rather, $S$ is $\lambda x.True$ and hence trivial in this example. Moreover, in the example, $R$ is $\alpha \Rightarrow bool$, and any term $r$ of type $R$ defines a set whose elements are of type $\alpha$; $Abs_{set}\ r$ is that set.

# $Collect$

We have seen $Collect$ before in the theory file exercise_03 (naïve set theory).

$Collect\, f$ is the set whose characteristic function is $f$. The usual concrete syntax is $\{x \mid f\, x\}$. The construct is called set comprehension.

Note also that $Collect$ is the same as $Abs_{set}$ here, so there is no need to have them as separate constants, and for this reason Isabelle theory file Set.thy only provides $Collect$.

# The $\in$-Sign

We define

$$x \in A \;\;=\;\; (Rep_{set}\ A)\ x$$

Since $Rep_{set}$ has type $\alpha\, set \Rightarrow (\alpha \Rightarrow bool)$, this means that $x$ is of type $\alpha$ and $A$ is of type $(\alpha \Rightarrow bool)$. Therefore $\in$ is of type $\alpha \Rightarrow (\alpha\, set) \Rightarrow bool$ (but written infix).

In the the Isabelle theory Set.thy, you will indeed find that the constant op : (Isabelle syntax for $\in$) has type $[\alpha, \alpha\, set] \Rightarrow bool$. However, you will not find anything directly corresponding to $Rep_{set}$.

One can see that this setup is equivalent to the one we have here (which was presented like that for the sake of generality). There are two axioms in Set.thy:

**axioms**

mem_Collect_eq [ iff ]: "(a : {x. P(x)}) = P(a)"

Collect_mem_eq [simp]: "{x. x:A} = A"

These axioms can be translated into definitions as follows:

$$
\begin{aligned}
& a \in \{x \mid P\,x\} = P\,a \leadsto \\
& a \in (Collect\,P) = P\,a \leadsto \\
& a \in (Abs_{set}\,P) = P\,a \leadsto \\
& Rep_{set}(Abs_{set}\,P)\,a = P\,a \leadsto Rep_{set}(Abs_{set}\,P) = P
\end{aligned}
$$

The last step uses extensionality.

Now the second one:

$$
\begin{aligned}
& \{x \mid x \in A\} = A \leadsto \\
& \{x \mid (Rep_{set}A)\,x\} = A \leadsto \\
& Collect(Rep_{set}A) = A
\end{aligned}
$$

Ignoring some universal quantifications (these are implicit in Isabelle), these are the isomorphy axioms for $set$.

# Consistent Set Theory

Typed set theory is a conservative extension of HOL and hence consistent.

Recall the problems with untyped set theory.

# “Exactly one” Term

When we say that there is “exactly one” $f$, this is meant modulo equality in HOL. This means that e.g. $\lambda x :: \alpha\, y :: \beta . y = b \wedge x = a$ is also such a term since $(\lambda x :: \alpha\, y :: \beta . x = a \wedge y = b) = (\lambda x :: \alpha y :: \beta .\, y = b \wedge x = a)$ is derivable in HOL.

# $Rep_\times$

$Rep_\times$ would be the generic name for one of the two isomorphism-defining functions.

Since $Rep_\times$ can not be represented directly for lexical reasons, type definitions in Isabelle provide two names for a type, one if the type is used as such, and one for the purpose of generating the names of the isomorphism-defining functions.

# Iteration of $\lambda$'s

We write $\lambda a :: \alpha\ b :: \beta.\ \lambda x :: \alpha\ y :: \beta.\ x = a \wedge y = b$ rather than $\lambda a :: \alpha\ b :: \beta\ x :: \alpha\ y :: \beta. x = a \wedge y = b$ to emphasize the idea that one first applies $Pair\_Rep$ to $a$ and $b$, and the result is a function representing a pair, wich can then be applied to $x$ and $y$.

# Sum Types

Idea of sum or union type: $t$ is in the sum of $\tau$ and $\sigma$ if $t$ is either in $\tau$ or in $\sigma$. To do this formally in our type system, and also in the type system of functional programming languages like ML, $t$ must be wrapped to signal if it is of type $\tau$ or of type $\sigma$.

For example, in ML one could define

$$\texttt{datatype}\ (\alpha, \beta)\ \texttt{sum} = \mathit{Inl}\ \alpha \mid \mathit{Inr}\ \beta$$

So an element of $(\alpha, \beta)$ sum is either $\mathit{Inl}\ a$ where $a :: \alpha$ or $\mathit{Inr}\ b$ where $b :: \beta$.

# Defining `even`

Suppose we have a type `nat` and a constant $+$ with the expected meaning. We want to define a type `even` of even numbers. What is an even number?

The following choice of $S$ is adequate:

$$S \equiv \lambda x.\, \exists n.\, x = n + n$$

Using the Isabelle scheme, this would be

**typedef** (Even)
    even = "{x. ∃ y.x=y+y}"

We could then go on by defining an operation `PLUS` on `even`, say as follows:

**constdefs**

PLUS::[even,even] $\rightarrow$ even ( **infixl** 56)
PLUS_def "op PLUS $\equiv\lambda$xy. Abs_Even(Rep_Even(x)+Rep_Even(x))"

Note that we chose to use names `even` and `Even`, but we could have used the same name twice as well.

# References

[GM93] Michael J. C. Gordon and Tom F. Melham, editors. *Introduction to HOL*. Cambridge University Press, 1993.