Chapter 3

HOL:Foundations

Introduction

- Stands for Higher Order Logic
- Denotes both a logic and a system
- Logic is an evolution of Alonzo Church's Simple Theory of Types (1940)
- System is an evolution of LCF (1979)
- Intent of this lecture: give an overview of HOL

Some Logical History

- Frege was a logicist (math is a subset of logic)
- Proposed a system on which (he thought) all mathematics could be derived (in principle)
- Bertrand Russell found paradox in Frege's system
- Proposed the Ramified Theory of Types
- Wrote *Principia Mathematica* with Whitehead
- An attempt at developing basic mathematics completely formally

"My intellect never recovered from the strain"

Russell's Paradox

Definition

A set s does not contain itself if $s \notin s$

Fact

Consider $X = \{s \mid s \notin s\}$. X is the set of all sets that do not contain themselves.

- *If $X \in X$ then X does not contain itself, i.e., $X \notin X$*
- *If $X \notin X$ then X contains itself, i.e., $X \in X$*

So $X \in X$ iff $X \notin X$. Contradiction.

- Gottlob, we have a problem!

Type Theory

- Problem: even allowing the expression of the notion of sets that do not contain themselves leads to contradiction
- One solution: ban such self-referential expressions (so-called *vicious circles*)
- Russell's proposal: invent a hierarchy of types
- Elements of lower types could not be applied to elements of higher types
- Blocks the paradox because $X \in X$ no longer a well-formed expression

Type Theories

- Russell's Ramified Theory of Types was very complex
- Simplified by Frank Ramsey in 1920s
- A. Church used typed λ-calculus to give a sleek presentation (Simple Theory of Types 1940)
- An earlier attempt by Church used untyped λ-calculus as a foundation for mathematics. It was inconsistent.
- HOL is a version of Church's 1940 logic.
- Many other variants as well, *e.g.*, Calculus of Constructions

History of HOL Implementations

- Late 1960’s : Dana Scott’s Domain Theory
- Logic of Computable Functions: a (first order) logic for Scott’s theory
- Implemented in Edinburgh LCF (mid-1970s)
- Early 1980’s : Mike Gordon swapped Scott’s logic for Church’s
- Kept much of LCF implementation

Contemporary Implementations of HOL

- HOL-Light (Harrison)
- HOL-4 (Gordon, Slind, Norrish, others)
- Isabelle/HOL (Paulson,Nipkow)
- ProofPower (Arthan)
- reFLect (Intel)

Related systems:

- PVS (extension of Church's logic with dependent types and subtypes)
- ACL2 (built on Common Lisp subset)
- MIZAR (Tarski-Grothendieck set theory)

Page of Logic Implementations

For a collection of logic implementations see

`http://www.cs.ru.nl/~freek/digimath/index.html`

Motivation

- Higher-order logic (HOL) is an expressive foundation for
  mathematics: analysis, algebra, . . .
  computer science: program correctness, hardware verification, . . .
- Reasoning in HOL is classical.
- Still important: modeling of problems (now in HOL).
- Still important: deriving relevant reasoning principles.

(rev. 12275)

Motivation (2)

- HOL offers safety through strength:
  - small kernel of constants and axioms;
  - Safety via conservative (definitional) extensions.
- Contrast with
  - weak logics (e.g., propositional logic): can't define much;
  - axiomatic extensions: can lead to inconsistency

Bertrand Russell once likened the advantages of postulation over definition to the advantages of theft over honest toil!

Alternatives to Isabelle/HOL

- We will use and focus on Isabelle/HOL.
- Could forgo the use of a meta-logic and employ alternatives, e.g., HOL system or PVS. Or use constructive alternatives such as Coq or Nuprl.
- Choice depends on culture and application.

Which Foundation?

- Set theory is often seen as the basis for mathematics.
  - Zermelo-Fraenkel, Bernays-Gödel, . . .
  - Set theories (both) distinguish between sets and classes.
  - Consistency maintained as some collections are "too big" to be sets, e.g., class of all sets is not a set. A class cannot belong to another class (let alone a set)!
- HOL as an alternative (Church 1940, Henkin 1950).
  - **Rationale:** one usually works with typed entities.
  - Isabelle/HOL also supports like polymorphism and type classes.
    HOL is weaker than ZF set theory, but for most applications this does not matter. If you prefer ML to Lisp, you will probably prefer HOL to ZF. —Larry Paulson
- Another alternative: category theory (Eilenberg, Mac Lane)

(rev. 12275)

Meaning of "Higher Order"

1st-order: quantification over individuals (0th-order objects).

$$\forall x,\, y.\, R(x,y) \longrightarrow R(y,x)$$

2nd-order: quantification over predicates and functions.

$$\begin{array}{rcl} false & \equiv & \forall P.\, P \\ P \wedge Q & \equiv & \forall R.\, (P \longrightarrow Q \longrightarrow R) \longrightarrow R \end{array}$$

3rd-order: quantify over variables whose arguments are predicates.

⋮

"higher order" $\leftrightsquigarrow$ union of all finite orders

(rev. 12275)

Basic HOL Syntax (1)

- Types:

$$\tau \ ::= \ bool \mid ind \mid \tau \Rightarrow \tau$$

  - $bool$ and ind are also called o and i in literature [Chu40, And86]
  - Isabelle allows definitions of new type constructors, e.g., $list(bool)$
  - Isabelle supports polymorphic type definitions, e.g., $list(\alpha)$

- Terms: ($\mathcal{V}$ set of variables and $\mathcal{C}$ set of constants)

$$\mathcal{T} \ ::= \ \mathcal{V} \mid \mathcal{C} \mid (\mathcal{T}\mathcal{T}) \mid \lambda\mathcal{V}.\mathcal{T}$$

  - Terms are simply-typed.
  - Terms of type $bool$ are called (well-formed) formulae.

Basic HOL Syntax (2)

- Constants are always supplied with types and include:

  $True, False : bool$

  $\_ = \_ : \tau \Rightarrow \tau \Rightarrow bool$ (for all types τ)

  $\_ \longrightarrow \_ : bool \Rightarrow bool \Rightarrow bool$

  $\iota\_ : (\tau \Rightarrow bool) \Rightarrow \tau$ (for all types τ)

- Note that the description operator ιf yields the unique element x for which $f\ x$ is $True$, provided it exists. Otherwise, it yields an arbitrary value.
- Note that in Isabelle, the provisos “for all types τ” can be expressed by using polymorphic type variables α.

HOL Semantics

- Intuitively an extension of many-sorted semantics with functions
  - FOL: structure is domain and functions/relations
$$\langle \mathcal{D}, (f_i)_{i \in F}, (r_i)_{i \in R} \rangle$$
  - Many-sorted FOL: domains are sort-indexed
$$\langle (\mathcal{D}_i)_{i \in S}, (f_i)_{i \in F}, (r_i)_{i \in R} \rangle$$
  - HOL extends idea: domain $\mathcal{D}$ is indexed by (infinitely many) types
- Our presentation ignores polymorphism on the object-logical level, it is treated on the meta-level, though (a version covering object-level parametric polymorphism is [GM93]).

Model Based on Universe of Sets $\mathcal{U}$

Definition 1 (Universe):

$\mathcal{U}$ is a collection of sets, fulfilling closure conditions:

Inhab: Each $X \in \mathcal{U}$ is a nonempty set

Sub: If $X \in \mathcal{U}$ and $Y \neq \emptyset \subseteq X$, then $Y \in \mathcal{U}$

Prod: If $X, Y \in \mathcal{U}$ then $X \times Y \in \mathcal{U}$.

$X \times Y$ is Cartesian product, $\{\{x\}, \{x, y\}\}$ encodes (x, y)

Pow: If $X \in \mathcal{U}$ then $\mathcal{P}(X) = \{Y : Y \subseteq X\} \in \mathcal{U}$

Infty: $\mathcal{U}$ contains a distinguished infinite set I

(rev. 12275)

Universe of Sets $\mathcal{U}$ (cont.)

- **Function space:**
  $X \Rightarrow Y$ is the set of (graphs of all total) functions from X to Y
  - For X and Y nonempty, $X \Rightarrow Y$ is a nonempty subset of $\mathcal{P}(X \times Y)$
  - From closure conditions: $X, Y \in \mathcal{U}$ then so is $X \Rightarrow Y$.
- **Distinguished sets:**
  from **Infty** and **Sub** there is (at least one) set
  Unit: A distinguished 1 element set $\{1\}$
  Bool: A distinguished 2 element set $\{T, F\}$.

(rev. 12275)

Definition 2 (Frame):

A frame is a collection $(\mathcal{D}_\alpha)_{\alpha \in \tau}$ with $\mathcal{D}_\alpha \in \mathcal{U}$, for $\alpha \in \tau$ and

- $\mathcal{D}_{bool} = \{T, F\}$
- $\mathcal{D}_{ind} = X$ where X is some infinite set of individuals
- $\mathcal{D}_{\alpha \Rightarrow \beta} \subseteq \mathcal{D}_\alpha \Rightarrow \mathcal{D}_\beta$, i.e., some collection of functions from D_α to D_β

Example: $\mathcal{D}_{bool \Rightarrow bool}$ is some nonempty subset of functions from $\{T, F\}$ to $\{T, F\}$. Some of these subsets contain, e.g., the identity function, others do not.

Definition 3 (Interpretation):

An interpretation $\langle(\mathcal{D}_\alpha)_{\alpha\in\tau}, \mathcal{J}\rangle$ consists of a frame $(\mathcal{D}_\alpha)_{\alpha\in\tau}$ and a denotation function $\mathcal{J}$ mapping each constant of type α to an element of $\mathcal{D}_\alpha$ where:

- $\mathcal{J}(\mathit{True}) = T$ and $\mathcal{J}(\mathit{False}) = F$
- $\mathcal{J}(=_{\alpha\Rightarrow\alpha\Rightarrow bool})$ is the identity on $\mathcal{D}_\alpha$
- $\mathcal{J}(\longrightarrow)$ denotes the implication function over $\mathcal{D}_{bool}$, i.e.,

$$b \to b' = \begin{cases} F & \text{if } b = T \text{ and } b' = F \\ T & \text{otherwise} \end{cases}$$

- $\mathcal{J}(\iota_{(\alpha\Rightarrow bool)\Rightarrow\alpha}) \in (\mathcal{D}_\alpha \Rightarrow \mathcal{D}_{bool}) \Rightarrow \mathcal{D}_\alpha$ denotes the function

$$\mathit{the}(f) = \begin{cases} a & \text{if } f = (\lambda x.x = a) \\ y & \text{otherwise } (y \in \mathcal{D}_\alpha \text{ is arbitrary}) \end{cases}$$

Definition 4 (Generalized Models):

An interpretation $\mathfrak{M} = \langle (\mathcal{D}_\alpha)_{\alpha\in\tau}, \mathcal{J} \rangle$ is a (general) model for HOL iff there is a binary function $\mathcal{V}^{\mathfrak{M}}$ such that

- for all type-indexed families of substitutions $\sigma = (\sigma_\alpha)_{\alpha\in\tau}$ and terms t of type α, $\mathcal{V}^{\mathfrak{M}}(\sigma, t) \in \mathcal{D}_\alpha$, and
- for all type-indexed families of substitutions $\sigma = (\sigma_\alpha)_{\alpha\in\tau}$,
  - (a) $\mathcal{V}^{\mathfrak{M}}(\sigma, x_\alpha) = \sigma_\alpha(x_\alpha)$
  - (b) $\mathcal{V}^{\mathfrak{M}}(\sigma, c) = \mathcal{J}(c)$, for c a (primitive) constant
  - (c) $\mathcal{V}^{\mathfrak{M}}(\sigma, s_{\alpha\Rightarrow\beta}t_\alpha) = \mathcal{V}^{\mathfrak{M}}(\sigma, s)\mathcal{V}^{\mathfrak{M}}(\sigma, t)$
    i.e., the value of the function $\mathcal{V}^{\mathfrak{M}}(\sigma, s)$ at the argument $\mathcal{V}^{\mathfrak{M}}(\sigma, t)$
  - (d) $\mathcal{V}^{\mathfrak{M}}(\lambda x_\alpha . t_\beta) =$ "the function from $\mathcal{D}_\alpha$ into $\mathcal{D}_\beta$ whose value for each $z \in \mathcal{D}_\alpha$ is $\mathcal{V}^{\mathfrak{M}}(\sigma[x \leftarrow z], t)$"

(rev. 12275)

Generalized Models - Facts (1)

- **If** $\mathfrak{M}$ is a general model and σ a substitution, **then** $\mathcal{V}^{\mathfrak{M}}(\sigma, t)$ is uniquely determined, for every term t.
  $\mathcal{V}^{\mathfrak{M}}(\sigma, t)$ is value of t in $\mathfrak{M}$ w.r.t. σ.
- Gives rise to the standard notion of satisfiability/validity:
  - We write $\mathcal{V}^{\mathfrak{M}}, \sigma \models \phi$ for $\mathcal{V}^{\mathfrak{M}}(\sigma, \phi) = T$.
  - ϕ is satisfiable in $\mathfrak{M}$ if $\mathcal{V}^{\mathfrak{M}}, \sigma \models \phi$, for some substitution σ.
  - ϕ is valid in $\mathfrak{M}$ if $\mathcal{V}^{\mathfrak{M}}, \sigma \models \phi$, for every substitution σ.
  - ϕ is valid (in the general sense) if ϕ is valid in every general model $\mathfrak{M}$.

Generalized Models - Facts (2)

- Not all interpretations are general models.
- Closure conditions guarantee every well-formed formula has a value under every assignment, e.g.,

closure under functions: identity function from $\mathcal{D}_\alpha$ to $\mathcal{D}_\alpha$ must belong to $\mathcal{D}_{\alpha \Rightarrow \alpha}$ so that $\mathcal{V}^{\mathfrak{M}}(\sigma, \lambda x_\alpha.\, x)$ is defined.

closure under application:

- if $\mathcal{D}_N$ is set of natural numbers and
- $\mathcal{D}_{N \Rightarrow N \Rightarrow N}$ contains addition function p where $p\,x\,y = x + y$
- then $\mathcal{D}_{N \Rightarrow N}$ must contain $k\,x = 2x + 5$ since $k = \mathcal{V}^{\mathfrak{M}}(\sigma, \lambda x.\, f\,(f\,x\,x)\,y)$ where $\sigma(f) = p$ and $\sigma(y) = 5$.

(rev. 12275)

Standard Models

Definition 5 (Standard Models):

A general model is a standard model iff for all $\alpha, \beta \in \tau$, $\mathcal{D}_{\alpha \Rightarrow \beta}$ is the set of all functions from $\mathcal{D}_\alpha$ to $\mathcal{D}_\beta$.

- A standard model is a general model, but not necessary vice versa.
- Analogous definitions for satisfiability and validity w.r.t. standard models.

Standard Models

Definition 5 (Standard Models):

A general model is a standard model iff for all $\alpha, \beta \in \tau$, $\mathcal{D}_{\alpha \Rightarrow \beta}$ is the set of all functions from $\mathcal{D}_\alpha$ to $\mathcal{D}_\beta$.

- A standard model is a general model, but not necessary vice versa.
- Analogous definitions for satisfiability and validity w.r.t. standard models.
- We can now re-introduce HOL in Isabelle's meta-logic.

Isabelle/HOL

The syntax of the core-language is introduced by:

consts

Not	::	bool $\Rightarrow$ bool	("$\neg$ _" [40] 40)
True	::	bool	
False	::	bool	
If	::	[bool, 'a, 'a] $\Rightarrow$ 'a	("(if _ then _ else _)")
The	::	('a $\Rightarrow$ bool) $\Rightarrow$ 'a	(**binder** "THE " 10)
All	::	('a $\Rightarrow$ bool) $\Rightarrow$ bool	(**binder** "$\forall$ " 10)
Ex	::	('a $\Rightarrow$ bool) $\Rightarrow$ bool	(**binder** "$\exists$ " 10)
$=$	::	['a, 'a] $\Rightarrow$ bool	(**infixl** 50)
$\wedge$	::	[bool, bool] $\Rightarrow$ bool	(infixr 35)
$\vee$	::	[bool, bool] $\Rightarrow$ bool	(infixr 30)
$\longrightarrow$	::	[bool, bool] $\Rightarrow$ bool	(infixr 25)

(rev. 12275)

The Axioms of HOL (1)

axioms

refl : "t = t"
subst: "⟦ s = t; P(s) ⟧ ⟹ P(t)"

ext: "(⋀x. f x = g x) ⟹ (λx. f x) = (λx. g x)"

impl: "(P ⟹ Q) ⟹ P⟶Q"
mp: "⟦ P⟶Q; P ⟧ ⟹ Q"

iff : "(P⟶Q) ⟶(Q⟶P) ⟶(P=Q)"
True_or_False : "(P=True) ∨(P=False)"

the_eq_trivial : "(THE x. x = a) = (a::'a)"

(rev. 12275)

The Axioms of HOL (2)

Additionally, there is:

- universal α, β, and η congruence on terms (implicitly),
- the axiom of infinity, and
- the axiom of choice (Hilbert operator).
- This is the entire basis!

Core Definitions of HOL

defs

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	$\equiv THE\ z::'a.\ (P=True \longrightarrow z=x) \wedge (P=False \longrightarrow z=y)$

(rev. 12275)

Meta-theoretic Properties of HOL

Theorem 1 (Soundness of HOL, [And86]):

HOL is sound w.r.t. to general models.

$$\vdash_{HOL} \phi \quad \text{implies} \quad \phi \text{ is valid}$$

Theorem 2 (Completeness of HOL, [And86]):

- HOL is complete w.r.t. to general models.

$$\phi \text{ is valid} \quad \text{implies} \quad \vdash_{HOL} \phi$$

- HOL is complete w.r.t. to standard models.

Theorem 3 (HOL with infinity, [And86]):

- HOL+infinity is complete w.r.t. general models.
- HOL+infinity is incomplete w.r.t. standard models.

(rev. 12275)

Conclusions

- HOL generalizes semantics of FOL
  - *bool* serves as type of propositions
  - Syntax/semantics allows for higher-order functions
- Logic is rather minimal: 8 rules, more-or-less obvious
- Logic is very powerful in terms of what we can represent/derive.
  - Other "logical" syntax
  - Rich theories via conservative extensions (topic for next few weeks!)

(rev. 12275)

Bibliography

- M. J. C. Gordon and T. F. Melham, Introduction to HOL: A theorem proving environment for higher order logic, Cambridge University Press, 1993.
- Peter B. Andrews, An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof, Academic Press, 1986.
- Tobias Nipkow and Lawrence C. Paulson and Markus Wenzel, Isabelle/HOL — A Proof Assistant for Higher-Order Logic, Springer-Verlag, LNCS 2283, 2002.

References

[Acz77] Peter Aczel. *Handbook of Mathematical Logic*, chapter An Introduction to Inductive Definitions, pages 739–782. North-Holland, 1977.

[And86] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proofs*. Academic Press, 1986.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and* All That. Cambridge University Press, 1998.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schliessen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in [Sza69].

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1989.

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

[HHPW96] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philipp Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.

[Höl90] Steffen Hölldobler. Conditional equational theories and complete sets of transformations. *Theoretical Computer Science*, 75(1&2):85–110, 1990.

[Klo93] Jan Willem Klop. *Handbook of Logic in Computer Science*, chapter "Term Rewriting Systems". Oxford: Clarendon Press, 1993.

[LP81] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1981.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[Nip93] Tobias Nipkow. *Logical Environments*, chapter Order-Sorted Polymorphism in Isabelle, pages 164–188. Cambridge University Press, 1993.

[NN99] Wolfgang Naraschewski and Tobias Nipkow. Type inference verified: Algorithm $\mathcal{W}$ in isabelle/hol. *Journal of Automated Reasoning*, 23(3-4):299–318, 1999.

[Pau96] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996.

[Pau03] Lawrence C. Paulson. *The Isabelle Reference Manual*. Computer Laboratory, University of Cambridge, March 2003.

[PM68] Dag Prawitz and Per-Erik Malmnäs. A survey of some connections between classical, intuitionistic and minimal logic. In A. Schmidt and H. Schütte, editors, *Contributions to Mathematical Logic*, pages 215–229. North-Holland, 1968.

[Pra65] Dag Prawitz. *Natural Deduction: A proof theoretical study*. Almqvist and Wiksell, 1965.

[Sza69] M. E. Szabo. *The Collected Papers of Gerhard Gentzen*. North-Holland, 1969.

[Tho95] Simon Thompson. *Miranda: The Craft of Functional Programming*. Addison-Wesley, 1995.

[Tho99] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1999. Second Edition.

[vD80] Dirk van Dalen. *Logic and Structure*. Springer-Verlag, 1980. An introductory textbook on logic.

[Vel94] Daniel J. Velleman. *How to Prove It*. Cambridge University Press, 1994.

[vH67] Jean van Heijenoort, editor. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-193*. Harvard University Press, 1967. Contains translations of original works by David Hilbert.

[WB89] Phillip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.

[WR25] Alfred N. Whitehead and Bertrand Russell. *Principia Mathematica*, volume 1. Cambridge University Press, 1925. 2nd edition.

Higher-order Logic: Conservative Extensions

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.

(rev. 32934)

Conservative Theory Extensions: Basics

Terminology and basic definitions (c.f. [GM93]):

Definition 6 (theory):

A (syntactic) theory T is a triple (χ, Σ, A), where χ is a type signature, Σ a signature, and A a set of axioms.

Definition 7 (consistent):

A theory T is consistent iff $False$ is not provable in T.

Definition 8 (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 $A \subseteq A'$.

(rev. 32934)

Definitions (Cont.)

Definition 9 (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 $|_{\Sigma}$ filters away all formulas not belonging to Σ.

Counterexample:

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

(rev. 32934)

Consistency Preserved

Lemma 1 (consistency):

If T' is a conservative extension of a consistent theory T, then

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

(rev. 32934)

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 10 (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.

(rev. 32934)

Constant Definitions are Conservative

Lemma 2 (constant definitions):

A constant definition is a conservative extension.

Proof Sketch:

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

Side Conditions

Where are those side conditions needed? What goes wrong?

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

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

Reconsider the definition of True.

(rev. 32934)

Constant Definition: Examples

Definitions of $True$, $False$, ¬, ∧, ∨, ∀, and ∃ 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:	$P \vee Q$	$\equiv \forall R.\ (P \longrightarrow R) \longrightarrow (Q \longrightarrow R) \longrightarrow R$

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

(rev. 32934)

More Constant Definitions in Isabelle

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

consts

```
Let  ::  ['a, 'a ⇒ 'b] ⇒ 'b
If   ::  [bool, 'a, 'a] ⇒ 'a
Ex1  ::  ('a ⇒ bool) ⇒ bool
```

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)"
```

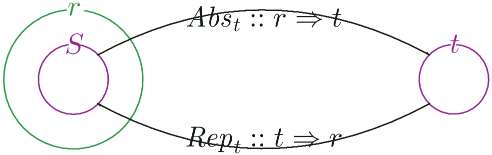
Note: ⇒ is function type arrow; recall syntax for [...] ⇒ ...

(rev. 32934)

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.



(rev. 32934)

Type Definition: Definition

Definition 11 (type definition):

Assume a theory $T = (\chi, \Sigma, A)$ and a type r and a term S of type $r \Rightarrow bool$.

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

$$\begin{array}{rcccl} \chi' & = & \chi & \uplus & \{t\}, \\ \Sigma' & = & \Sigma & \cup & \{Abs_t :: r \Rightarrow t, Rep_t :: t \Rightarrow r\} \\ A' & = & A & \cup & \{\forall x. Abs_t(Rep_t\, x) = x, \\ & & & & \quad \forall x. S\, x \longrightarrow Rep_t(Abs_t\, x) = x\} \end{array}$$

Proof obligation $T \vdash \exists x.\, S\, x$ (inside HOL)

(rev. 32934)

Type Definitions are Conservative

Lemma 3 (type definitions):

A type definition is a conservative extension.

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 $r \equiv \alpha \Rightarrow bool$ (α is any type variable), $t \equiv \alpha\ set$ (or set), $S \equiv \lambda x :: \alpha \Rightarrow bool.\ True$

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

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

(rev. 32934)

Sets: Remarks

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

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

(rev. 32934)

More Constants for Sets

For convenient use of sets, we define more constants:

$$
\begin{aligned}
\{x \mid f\,x\} &\cong \mathit{Collect}\ f = \mathit{Abs}_{set}\ f \\
x \in A &= (\mathit{Rep}_{set}\ A)\ x \\
A \cup B &= \{x \mid x \in A \lor x \in B\} \\
&\vdots
\end{aligned}
$$

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.

(rev. 32934)

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, $\lambda x :: \alpha.\ \lambda y :: \beta.\ x = a \wedge y = b$).
- Some functions of type $\alpha \Rightarrow \beta \Rightarrow bool$ represent pairs and others don't (e.g., the function $\lambda x.\ \lambda y.\ True$ does not represent a pair). The ones that do are are equal to $\lambda x :: \alpha.\ \lambda y :: \beta.\ x = a \wedge y = b$, for some a and b.

Type Definition for Pairs

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

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

It is convenient to define a constant Pair_Rep (not to be confused with $Rep_{\times}$) as follows:
Pair_Rep a b = λx ::' a. λ y ::' b. x=a ∧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 (α, β) `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.

(rev. 32934)

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) "+" (infixr 10)
   = "{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 presented a method to safely build up larger theories:
  - Constant definitions;
  - Type definitions.
- Subtle side conditions.
- A new type must be isomorphic to a “subset” of an existing type.

(rev. 32934)

More Detailed Explanations

(rev. 32934)

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.

(rev. 32934)

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.

(rev. 32934)

Closed Terms

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

(rev. 32934)

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.

(rev. 32934)

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 λ-bound, and so the type of those variables is given by the type superscripts.

(rev. 32934)

Domains of Σ, Γ

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

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

Note the slight abuse of notation.

(rev. 32934)

constdefs

In Isabelle theory files, `consts` is the keyword preceding a sequence of constant declarations (i.e., this is where the Σ 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 $id \equiv \lambda x.x$.

(rev. 32934)

$$S$$

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

(rev. 32934)

Fresh t

The type constructor t must not occur in χ.

What Is t?

We use the letter χ 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).

(rev. 32934)

More Detailed Explanations 726

$\uplus$

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.

(rev. 32934)

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 u of type t for which $S\,u$ holds.

Isomorphism

The formulas

$$\forall x. Abs_t(Rep_t\, x) = x$$
$$\forall x. S\, x \longrightarrow Rep_t(Abs_t\, x) = x$$

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.

(rev. 32934)

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 f of type r defines a set whose elements are of type α; $Abs_{set}\ f$ is that set.

(rev. 32934)

$Collect$

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 α 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)"

(rev. 32934)

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

These axioms can be translated into definitions as follows:

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

The last step uses extensionality.

Now the second one:

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

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.

(rev. 32934)

$Rep_\times$

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

Since $Rep_\times$ cannot 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.

(rev. 32934)

Iteration of λ'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 τ and σ if t is either in τ or in σ. 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 τ or of type σ.

For example, in ML one could define

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

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

(rev. 32934)

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?

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

More Detailed Explanations 742

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.

(rev. 32934)