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/

Isabelle's Metalogic and Proof Objects

Burkhart Wolff

Overview

This chapter reconsiders again Isabelle as a logical framework. This involves:

Overview

This chapter reconsiders again Isabelle as a logical framework. This involves:

- its version of a typed λ-calculus

Overview

This chapter reconsiders again Isabelle as a logical framework. This involves:

- its version of a typed λ-calculus
- its elementary logic called Pure

Overview

This chapter reconsiders again Isabelle as a logical framework. This involves:

- its version of a typed λ-calculus
- its elementary logic called Pure
- a deeper understanding of *rule* / rtac etc,

Overview

This chapter reconsiders again Isabelle as a logical framework. This involves:

- its version of a typed λ-calculus
- its elementary logic called Pure
- a deeper understanding of *rule* / rtac etc,
- proof objects

Overview

This chapter reconsiders again Isabelle as a logical framework. This involves:

- its version of a typed λ-calculus
- its elementary logic called Pure
- a deeper understanding of *rule* / rtac etc,
- proof objects
- consequences

An Extension of the Typed λ-Calculus

Universal representation for object logics in Isabelle: A Typed λ-calculus extended by (parametric) polymorphism and type classes.

An Extension of the Typed λ-Calculus

Universal representation for object logics in Isabelle: A Typed λ-calculus extended by (parametric) polymorphism and type classes.

Historically, polymorphism in logics — although already used in the principia mathematica on the meta-level — is a fairly recent discovery (around 1975, first implementation: Edinburgh LCF). The consequences for Conservative Definitions have been sorted out in the early 80ies.

Polymorphism: Intuition

As in functional programming, the function $\_ = \_$ should be available on any type. This can be expressed by giving $\_ = \_$ the type $[\alpha, \alpha] \Rightarrow bool$ with α an explicit type variable as part of the type expression language.

Polymorphism: Intuition

As in functional programming, the function $\_ = \_$ should be available on any type. This can be expressed by giving $\_ = \_$ the type $[\alpha, \alpha] \Rightarrow bool$ with α an explicit type variable as part of the type expression language.

Adding type classes ("sorts of types") helps to separate universes of types from each other. $[\alpha :: term, \alpha] \Rightarrow bool$, for example, can be used to express that α may range over all types with individuals, but not predicates (i.e. $bool$ as in FOL).

Polymorphism: Intuition

As in functional programming, the function $\_ = \_$ should be available on any type. This can be expressed by giving $\_ = \_$ the type $[\alpha, \alpha] \Rightarrow bool$ with α an explicit type variable as part of the type expression language.

Adding type classes ("sorts of types") helps to separate universes of types from each other. $[\alpha :: term, \alpha] \Rightarrow bool$, for example, can be used to express that α may range over all types with individuals, but not predicates (i.e. $bool$ as in FOL).

Adding type constructors allows the introduction of $bool$, but also concepts such as $\alpha\, set$.

Polymorphism: Intuition

As in functional programming, the function $\_ = \_$ should be available on any type. This can be expressed by giving $\_ = \_$ the type $[\alpha, \alpha] \Rightarrow bool$ with α an explicit type variable as part of the type expression language.

Adding type classes ("sorts of types") helps to separate universes of types from each other. $[\alpha :: term, \alpha] \Rightarrow bool$, for example, can be used to express that α may range over all types with individuals, but not predicates (i.e. $bool$ as in FOL).

Adding type constructors allows the introduction of $bool$, but also concepts such as $\alpha\, set$.

We present a simplification of [NP95]. More formally, we have:

Syntax: Classes, Types, and Terms

Type classes (exemplary)

$$\kappa \quad ::= \quad ord \mid order \mid lattice \mid \ldots$$

Type constructors (exemplary)

$$\chi \quad ::= \quad bool \mid \_ \to \_ \mid ind \mid \_\ list \mid \_\ set \ldots$$

Polymorphic types

$$\tau \quad ::= \quad \alpha :: \{\kappa, \ldots, \kappa\} \mid (\tau, .., \tau)\chi \qquad (\alpha \text{ is type variable})$$

Raw terms (as before)

$$e \quad ::= \quad x \mid ?x \mid c \mid (ee) \mid (\lambda x^{\tau}.\, e)$$

ClaPolymorphic Type Inferences (1)

Prerequisites:

- a partial order $\leq$ on classes,
- . . . implying an equivalence on type class sets,
- a constant environment Σ, a variable environment Γ and a type environment ξ assigning to type variables (finite) sets of type classes,
- a type instance relation Δ assigning $(\kappa..\kappa)\chi$ to κ
- Type instances (denoted Θ) extend type environments to substitutions of types in terms,
- and two judgements $\Sigma, \xi \vdash \tau : \{\kappa..\kappa\}$ and $\Sigma, \Gamma, \xi \vdash e : \tau$

Polymorphic Type Inferences (2)

$$\frac{c : \tau \in \Sigma \quad \{\alpha_1 : S_1 \ldots \alpha_n : S_n\} \in \mathtt{tvc}(\tau) \quad (\Sigma, \xi \vdash \tau_i : S_i)_i}{\Gamma \vdash c : \tau[\alpha_1 := \tau_1, \ldots, \alpha_1 := \tau_n]} \; \mathtt{CONST}$$

$$\frac{}{\Sigma, \Gamma \vdash x : \Gamma(x)} \; \mathtt{ASM} \qquad \frac{}{\Sigma, \Gamma \vdash ?x : \Gamma(?x)} \; \mathtt{ASM}$$

$$\frac{\Sigma, \Gamma \vdash e : \sigma \to \tau \quad \Sigma, \Gamma \vdash e' : \sigma}{\Sigma, \Gamma \vdash e\, e' : \tau} \; \mathtt{APP} \qquad \frac{\Sigma, \Gamma[x : \sigma] \vdash e : \tau}{\Sigma, \Gamma \vdash \lambda x^{\sigma} .\, e \; : \sigma \to \tau} \; \mathtt{ABS}$$

`tvc` computes an assignment of all type variables occurring in τ to the set of all constraints associated to it in τ.

Polymorphic Type Inferences (3)

The second judgement $\Sigma, \xi \vdash \tau : \{\kappa..\kappa\}$ infers if a type is admissible to a class κ:

Polymorphic Type Inferences (3)

The second judgement $\Sigma, \xi \vdash \tau : \{\kappa..\kappa\}$ infers if a type is admissible to a class κ:

$$\frac{(\Sigma, \xi \vdash \tau : \kappa_i)_{i \in \{1 \ldots n\}}}{\Sigma, \xi \vdash \tau : \{\kappa_1, \ldots, \kappa_n\}} \qquad \frac{\Sigma, \xi \vdash \tau : \{\kappa_1, \ldots, \kappa_n\} \quad i \in \{1 \ldots n\}}{\Sigma, \xi \vdash \tau : \kappa_i}$$

$$\frac{\xi(\alpha) = S}{\Sigma, \xi \vdash \alpha : S} \qquad \frac{(\kappa_1, \ldots, \kappa_n)\chi \mapsto \kappa \in \Delta \quad (\Sigma, \xi \vdash \tau_i : \kappa_i)_{i \in \{1 \ldots n\}}}{(\Sigma, \xi \vdash (\tau_1, \ldots, \tau_n)\chi : \kappa}$$

$$\frac{\Sigma, \xi \vdash \tau : \kappa_1 \quad \kappa_1 \leq \kappa_2}{\Sigma, \xi \vdash \tau : \kappa_2}$$

Polymorphic Type Inferences (3)

The second judgement $\Sigma, \xi \vdash \tau : \{\kappa..\kappa\}$ infers if a type is admissible to a class κ:

$$\frac{(\Sigma, \xi \vdash \tau : \kappa_i)_{i \in \{1 \ldots n\}}}{\Sigma, \xi \vdash \tau : \{\kappa_1, \ldots, \kappa_n\}} \qquad \frac{\Sigma, \xi \vdash \tau : \{\kappa_1, \ldots, \kappa_n\} \quad i \in \{1 \ldots n\}}{\Sigma, \xi \vdash \tau : \kappa_i}$$

$$\frac{\xi(\alpha) = S}{\Sigma, \xi \vdash \alpha : S} \qquad \frac{(\kappa_1, \ldots, \kappa_n)\chi \mapsto \kappa \in \Delta \quad (\Sigma, \xi \vdash \tau_i : \kappa_i)_{i \in \{1 \ldots n\}}}{(\Sigma, \xi \vdash (\tau_1, \ldots, \tau_n)\chi : \kappa}$$

$$\frac{\Sigma, \xi \vdash \tau : \kappa_1 \quad \kappa_1 \leq \kappa_2}{\Sigma, \xi \vdash \tau : \kappa_2}$$

Note that there are constraints for Δ which are ommitted here (see [NP95] for details).

The Logic Pure

Tactics = Programs building Meta-Theorems

When constructing proofs, there are

Tactics = Programs building Meta-Theorems

When constructing proofs, there are

- logic specific aspects (its rules)

Tactics = Programs building Meta-Theorems

When constructing proofs, there are

- logic specific aspects (its rules)
- logic independent aspects such as:

Tactics = Programs building Meta-Theorems

When constructing proofs, there are

- logic specific aspects (its rules)
- logic independent aspects such as:
  - binding and substitution
  - typing
  - managing side-conditions
  - managing assumptions and their discharge

Tactics = Programs building Meta-Theorems

When constructing proofs, there are

- logic specific aspects (its rules)
- logic independent aspects such as:
  - binding and substitution
  - typing
  - managing side-conditions
  - managing assumptions and their discharge

In textbooks, the focus is typically on the former and the latter were only described in informal “provisos”.

Tactics = Programs building Meta-Theorems

When constructing proofs, there are

- logic specific aspects (its rules)
- logic independent aspects such as:
  - binding and substitution
  - typing
  - managing side-conditions
  - managing assumptions and their discharge

In textbooks, the focus is typically on the former and the latter were only described in informal "provisos".

- Using a metalogic Pure has two benefits:
  - shared implementations for the logic independent aspects, and
  - potential for "generic" proof procedures built on top of it.

Built on top of the syntactic language of the extended type class λ-calculus, Isabelle's meta-language Pure is implemented.

At least one type classes are assumed: $logic \in \kappa$. Moreover, at least two type constructors are assumed: $\mathtt{prop}, \_ \Rightarrow \_ \in \chi$.

Logic Based on $\lambda^{\rightarrow}$

Then the signature Σ of Pure is defined as follows:

Logic Based on $\lambda^{\rightarrow}$

Then the signature Σ of Pure is defined as follows:

- $\_ \Longrightarrow \_ : \mathtt{prop} \rightarrow \mathtt{prop} \rightarrow \mathtt{prop} \in \Sigma$,
- $\_ \equiv \_ :\alpha \rightarrow \alpha \rightarrow \mathtt{prop} \in \Sigma$, and
- $\bigwedge \_ : (\alpha \rightarrow \mathtt{prop}) \rightarrow \mathtt{prop} \in \Sigma$.

The _-notation is used to indicate infixes.

Terms of type $bool$ as in HOL, for example, were represented by a special constant $\mathtt{Trueprop} :: bool \Rightarrow \mathtt{prop}$. $\mathtt{Trueprop}\,\phi$ corresponds to the pr-operator in the "Propositional Logic in LF" encoding or the textbook notation "$\vdash \phi$".

(Trueprop is usually supressed syntactically.)

The Format of thm

Isabelle's Pure is

- implemented in the style of the LCF system: meta-level rules are SML functions on **thm**, possibly raising exceptions,

The Format of thm

Isabelle's Pure is

- implemented in the style of the LCF system: meta-level rules are SML functions on **thm**, possibly raising exceptions,
- uses natural deduction:
  each **thm** may depend on meta-level assumptions:

$$\phi[\phi, \ldots, \phi]$$

The Format of thm

Isabelle's Pure is

- implemented in the style of the LCF system: meta-level rules are SML functions on **thm**, possibly raising exceptions,
- uses natural deduction:
  each **thm** may depend on meta-level assumptions:

$$\phi[\phi, \ldots, \phi]$$

- each **thm** has a signature $(\Sigma, \chi, \kappa, \Delta)$.

Asumption and Rules for $\Rightarrow$

Manipulating meta-level assumptions:

$$\frac{}{\phi[\phi]}\ \texttt{assume} \qquad \frac{\begin{array}{c}[\phi]\\ \vdots\\ \psi\end{array}}{\phi \Rightarrow \psi}\ \Rightarrow\text{-}I \qquad \frac{\phi \Rightarrow \psi \quad \phi}{\psi}\ \Rightarrow\text{-}E$$

Note that $\Rightarrow$-*I* is now understood fully operationally: ϕ is erased from the meta-level assumption list of the premise of $\Rightarrow$-*I*.

Rules for $\equiv$: Equivalence Relation

Rules:

$$\frac{\phi \Rightarrow \psi \quad \psi \Rightarrow \phi}{\phi \equiv \psi} \equiv -I \qquad \frac{\phi \equiv \psi \quad \phi}{\psi} \equiv -E$$

$$\frac{}{a \equiv a} \equiv\text{-}\mathit{refl} \qquad \frac{a \equiv b}{b \equiv a} \equiv\text{-}\mathit{symm}$$

$$\frac{a \equiv b \quad b \equiv c}{a \equiv c} \equiv\text{-}\mathit{trans}$$

Rules for $\equiv$: λ (i.e., α, β, η) Conversions

Compare to $=_{\alpha,\beta,\eta}$.

$$\frac{}{(\lambda x.a) \equiv (\lambda y.a[x \leftarrow y])}\ \alpha^* \qquad \frac{}{(\lambda x.a)b \equiv (a[x \leftarrow b])}\ \beta$$

$$\frac{f \equiv g}{f\ x \equiv g\ x}\ \eta^{**}$$

Side condition $*$: y is not free in a.

Side condition $**$: x is not free in f, g and the meta-level asumptions.

Conversion is built into the proof system, and Isabelle routinely computes terms in α, β, η-normal-forms.

Note: These side conditions are directly implemented in the

SML code; in a way, this implemements similar side-conditions of object-logics once and for all.

Rules for $\equiv$: Abstraction, Combination

Rules

$$\frac{a \equiv b}{(\lambda x.a) \equiv (\lambda x.b)}\ \equiv\text{-}abstr^{*} \qquad \frac{f \equiv g \quad a \equiv b}{f\ a \equiv g\ b}\ \equiv\text{-}comb$$

Side condition $*$: x is not free in the meta-level assumptions.

Manipulating Meta-Variables

Rules:

$$\frac{\phi}{\phi[?x_1 := t_1, \ldots, ?x_n := t_n]}\ \texttt{instantiate}$$

`instantiate` can in fact also handle instantiations of type-meta variables, which we ignore throughout this presentation.

A somewhat exotic axiom scheme — traditionally treated as outside the core of Pure — introduces axiomatic type class invariants into the core logic:

$$\frac{}{\texttt{OFCLASS}(\alpha :: c, c\_class)}\ \texttt{class\_triv}$$

Rules for $\bigwedge$

Meta-quantification is formalized in higher-order abstract syntax: we write $\bigwedge x.\phi$ for $\bigwedge x.(\lambda x.\phi)$.

Rules for $\bigwedge$

Meta-quantification is formalized in higher-order abstract syntax: we write $\bigwedge x.\phi$ for $\bigwedge x.(\lambda x.\phi)$.

Rules:

$$\frac{\phi}{\bigwedge x.\phi}\ \bigwedge\text{-}I^{*} \qquad \frac{\bigwedge x.\phi}{\phi[x \leftarrow b]}\ \bigwedge\text{-}E$$

Rules for $\bigwedge$

Meta-quantification is formalized in higher-order abstract syntax: we write $\bigwedge x.\phi$ for $\bigwedge x.(\lambda x.\phi)$.

Rules:

$$\frac{\phi}{\bigwedge x.\phi}\ \bigwedge\text{-}I^{*} \qquad \frac{\bigwedge x.\phi}{\phi[x \leftarrow b]}\ \bigwedge\text{-}E$$

Side condition $*$: x is not free in meta-level assumptions. x may be a free variable or a meta-variable.

Rules for $\bigwedge$

Meta-quantification is formalized in higher-order abstract syntax: we write $\bigwedge x.\phi$ for $\bigwedge x.(\lambda x.\phi)$.

Rules:

$$\frac{\phi}{\bigwedge x.\phi}\ \bigwedge\text{-}I^* \qquad \frac{\bigwedge x.\phi}{\phi[x \leftarrow b]}\ \bigwedge\text{-}E$$

Side condition $*$: x is not free in meta-level assumptions. x may be a free variable or a meta-variable.

Note that combinations of $\bigwedge$-I^* and $\bigwedge$-E may therefore achieve the effect of replacing free variables by meta-variables.

What's different from HOL?

- no Falsum $\perp$,

What's different from HOL?

- no Falsum $\perp$,
- no classical :

What's different from HOL?

- no Falsum $\perp$,
- no classical :

  Pure is an intuitionistic fragment of HOL
- . . . what would be the consequences otherwise ?
- processes done by rtac / *rule* like:
  - lifting over assumptions
  - lifting over parameters

  are "rule schemes" implemented as tactical programs over Pure

Proof Objects

Although LCF-style systems were originally designed to avoid the construction of explicit proof-objects (as seen in LF), Isabelle has meanwhile a mechanism to “log” them during proof.

This has the following consequences:

- external proof-procedures can be used and recorded,
- proof-objects from extern provers may be imported,
- proof-objects of Isabelle can be checked externally.

How to generate Proof-Objects? (1)

theory ProofTest = Main:
ML{* *proofs := 2* *}

lemma a1 : " a ⟶a" **by**(auto)

ML{* *ProofSyntax.print_proof_of false (thm "a1");* *}

lemma a2 : " a ⟶b ⟶ a" **by**(auto)
ML{* *ProofSyntax.print_proof_of true (thm "a2");* *}

How to generate Proof-Objects? (2)

```
equal_elim · _ · _ •>
 (symmetric · _ · _ •>
   (combination · Trueprop · _ · _·_•>( reflexive · _)•>
     ( transitive · _ · _ · _ •>
       (? •> ( reflexive · _) •>
         (ΛH: _.
           equal_elim · _ · _ •>
            (symmetric · _ · _ •>
              (combination · _ · _ · _ · _ •>
                (combination · op ≡ · _· _· _•>( reflexive · _)•>
                  (Eq_TrueI · _ •> H)) •>
                ( reflexive · _))) •>
            ( reflexive · _))) •>
```

?))) ·>

Truel

How to generate Proof-Objects? (3)

```
equal_elim · _ · _ •>
 (symmetric · _ · _ •>
   (combination · Trueprop · _ · _ · _ •> ( reflexive · _) •>
     ( transitive · _ · _ · _ •>
       (? •> ( reflexive · _) •>
         (Λ H: _.
           equal_elim · _ · _ •>
            (symmetric · _ · _ •>
              (combination · _ · _ · _ · _ •>
                (combination · op≡·_·_·_•>( reflexive ·_)•>
                  ( transitive · _ · _ · _ •>
                    (? •> ( reflexive · _) •>
                      (Λ Ha: _.
```

```
                        equal_elim · _ · _ ·>
                         (symmetric · _ · _ ·>
                           (combination · _ · _ · _ · _ ·>
                             (combination · op≡·_·_·_·>
                               ( reflexive · _) ·>
                               (Eq_TrueI · _ ·> H)) ·>
                             ( reflexive · _))) ·>
                         ( reflexive · _))) ·>
                   ?)) ·>
              ( reflexive · _))) ·>
          ( reflexive · _))) ·>
     ?))) ·>
TrueI
```

How to generate Proof-Objects

The proof-checker:

ProofChecker.thm_of_proof thy prf

returns a **thm** for a valid proof!

How to generate Proof-Objects

The proof-checker:

ProofChecker.thm_of_proof thy prf

returns a **thm** for a valid proof!

It consists of 100 lines of code (although reusing ca. 1000 lines of kernel code).

Conclusion on Isabelle's Metalogic

Conclusion on Isabelle's Metalogic

- The logic Pure and its proof system are small,

Conclusion on Isabelle's Metalogic

- The logic Pure and its proof system are small,
- Even resolution, and d-resolution are not built-in; they are tactics over Pure,

Conclusion on Isabelle's Metalogic

- The logic Pure and its proof system are small,
- Even resolution, and d-resolution are not built-in; they are tactics over Pure,
- Isabelle can log proofs in proof objects,

Conclusion on Isabelle's Metalogic

- The logic Pure and its proof system are small,
- Even resolution, and d-resolution are not built-in; they are tactics over Pure,
- Isabelle can log proofs in proof objects,
- If you don't trust Isabelle, check proof-objects !!!

More Detailed Explanations

The names of $\Rightarrow$, $\equiv$, and $\bigwedge$

- $\Longrightarrow$ is called meta-implication,
- $\equiv$ is called meta-equality, and
- $\bigwedge$ is called meta-universal-quantification.

References

[NP95] Tobias Nipkow and Christian Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, 5(2):201–224, 1995.