Verificação Formal em Isabelle/HOL para Newbies

Alfio Martini

7 de novembro de 2019

Resumo

Isabelle/HOL é ao mesmo tempo um assistente de prova interativo e uma linguagem de alto nível para especificar e verificar formalmente qualquer sistema de computação que queiramos imaginar. Além disso, é também vastamente documentada com artigos de pesquisa, manuais e tutoriais. Neste caso específico, queremos abordar o essencial da linguagem para novatos que queiram aprender o conjunto mínimo de técnicas para construir provas por indução estrutural. Os pré-requisitos são mínimos: familiaridade com dedução natural e provas por indução estrutural. Esperamos estabelecer uma base para que o leitor possa se sentir confortável em ir adiante, caso suas ambições com Isabelle sejam maiores que os modestos objetivos traçados aqui.

Sumário

1 Introdução

Trabalhar com Isabelle significa construir teorias. De forma simplificada, uma **teoria** é uma coleçao de tipos, funções e teoremas da mesma forma que um módulo ou uma classe em uma típica linguagem de programação é uma coleção de atributos, métodos e algoritmos. O formato geral de uma teoria `T` é da seguinte forma:

theory `T`
imports $\texttt{B}_1 \ldots \texttt{B}_n$
begin
  declarações, definições e provas
end

onde $\texttt{B}_1 \ldots \texttt{B}_n$ são nomes de teorias existentes na qual `T` é baseada. Cada teoria `T` deve residir em um arquivo nomeado como `T.thy`.

A seguir, começamos apresentando uma teoria sobre números naturais. Construiremos a nossa própria versão deste tipo.

2 Uma Teoria para Números Naturais

Considere a seguinte declaração:

```
theory nat_tutorial
imports Main
        "HOL-Library.LaTeXsugar"
begin
```

A teoria `Main` é a união de todas as teorias pré-definidas de Isabelle. Sugerimos incluir `Main` sempre como o pai direto ou indireto de todas as teorias. A outra teoria que importamos, `LaTeXsugar`, é necessária para processar este documento, que foi produzido com o sistema de preparação de documentos do Isabelle, que é essencialmente um front-end para a tecnologia LaTeX tradicionalmente conhecida.

Isabelle é um assistente de prova genérico que permite que fórmulas sejam expressas em uma linguagem formal, mais notadamente um tipo de lógica de segunda ordem, que chamaremos daqui por diante de HOL. De forma simplificada, é suficiente que o leitor veja HOL através da seguinte equação:

$$\mathsf{HOL} = \texttt{Programação Funcional} + \texttt{Lógica}$$

Isto é, basicamente estamos extendendo a lógica de primeira ordem de forma a poder quantificar também sobre funções, conjuntos e relações. Além disso, Isabelle fornece uma grande quantidade de ferramentas para que possamos provar propriedades expressas por essas fórmulas. A principal aplicação deste assistente de prova é na área de verificação formal, a qual inclui a prova da correção de sistemas de hardware e software e na prova de propriedades de linguagens de programação e protocolos em geral. O estilo de apresentação que usamos aqui é fortemente inspirado e influenciado pelo ótimo [6].

Este tutorial está organizado da seguinte forma: na seção 2.1 apresentamos as definições básicas sobre uma representação sintática do conjuntos dos números naturais. Na seção 2.2 discutimos em detalhes a prova de associatividade da função de adição utilizando o estilo tradicional de scripts como sequência de comandos. Na seção 2.3, introduzimos brevemente a linguagem Isar, e reformulamos a mesma prova discutida anteriormente, mas agora através de uma linguagem de alto nível que permite impor estrutura sobre a prova e que captura com precisão o estilo tradicional que os cientistas da computação utilizam quando constroem tais demonstrações. Na última seção, retomamos rapidamente o que foi apresentado aqui e fazemos algumas sugestões de como o leitor pode prosseguir para continuar seus estudos nesta área fantástica e com essa ferramenta extremamente sofisticada e elegante.

2.1 Definições Básicas

Nesta seção, introduzimos um tipo de dado que pode ser visto como uma cópia isomórfica do conjuntos dos números naturais. Sobre os construtores deste tipo, podemos definir funções recursivas e provar várias propriedades que essas funções satisfazem. Uma vez que a definição do tipo de dado é uma construção indutiva, é natural que utilizemos a técnica de indução estrutural para verificação formal destas mesmas propriedades.

```
datatype Nat = Z | suc Nat
```

A declaração acima define um nome de tipo, neste caso Nat. Do lado direito do sinal de igualdade temos as expressões em HOL que definem os valores que habitam esse tipo. A cláusula Z define o caso base, isto é, que Z é um elemento do tipo Nat. A segunda cláusula, isto é, $suc\ Nat$ define o caso indutivo (recursivo). Essa expressão diz que ao aplicarmos o construtor (operador) suc a qualquer elemento do tipo Nat, temos um novo elemento do tipo Nat. Desta forma, o conjunto de todos os elementos (na realidade, **termos**) ou expressões que habitam esse tipo são:

$$\begin{array}{l} \hline Z \\ suc\ Z \\ suc\ (suc\ Z) \\ \vdots \\ suc^n\ Z \\ \vdots \\ \hline \end{array}$$

Desta forma, está implícito na declaração do **datatype** acima, que os construtores suc e Z possuem os seguintes tipos:

$$Z :: Nat \quad suc :: Nat \Rightarrow Nat$$

Toda declaração de um **datatype** em Isabelle gera (define) automaticamente uma regra de indução que diz o que é necessário para provar um propriedade sobre os valores do tipo. No caso acima, a regra de indução associada a declaração do **datatype** Nat acima é da seguinte forma:

Definição 2.1. *A seguinte regra de indução é automaticamente associada ao tipo Nat, após o Isabelle ter processado a definição do tipo via o comando **datatype**.*

$$\frac{\texttt{P Z} \qquad \bigwedge \texttt{x}_0.\ \dfrac{\texttt{P x}_0}{\texttt{P (suc x}_0\texttt{)}} \qquad}{\texttt{P n}}\ \mathsf{IndNat}$$

□

A regra acima, no cálculo de dedução natural é apresentada com uma condição adicional, isto é. a de que a variável $\texttt{x}_0$ deve ser arbitrária. Em outras palavras, $\texttt{x}_0$ não pode ocorrer livre nas hipóteses e premissas utilizadas para provar `P (suc x`$_0$`)`. A **meta-lógica** de Isabelle apresenta seu próprio quantificador universal ($\bigwedge$) para expressar esta noção de um valor arbitrário, o que evita que precisemos escrever essa condição adicional utilizando uma linguagem informal (e.g., português).
Desta forma, lendo a regra IndNat de baixo para cima, temos que, para provarmos uma propriedade P para qualquer elemento n do tipo Nat, precisamos fazer apenas duas coisas:

1. Provar que P vale para Z;
2. Assumindo que P vale para um elemento arbitrário x_0, provar que P também vale para $suc\ x_0$.

Considere a mesma regra para prova de indução do tipo de dado *Nat*, expressa no estilo de dedução natural sem o quantificador da meta-lógica $\bigwedge$, onde x_0 é uma variável arbitrária que não pode ocorrer **livre** em nenhuma hipótese utilizada para provar $P\ (suc\ x_0)$.

Definição 2.2. *Abaixo uma outra versão para a regra de indução sobre os elementos do tipo Nat, utilizando a apresentação tradicional do cálculo de dedução natural na linguagem da lógica de predicados. Note que as condições meta-lógicas são expressas agora informalmente (fora do formalismo da lógica de primeira ordem), isto é, em linguagem natural.*

Datatype Nat

$$\frac{\begin{array}{ccc} \vdots & \boxed{\begin{array}{c} x_0 :: Nat, P\ x_0 \\ \vdots \\ P\ (suc\ x_0) \end{array}} & (x_0\ local) \\ P\ Z & & \end{array}}{\forall n :: Nat.\ P\ n}\ \mathit{IndNat}$$

□

Exercício 2.1. *Compare as regras de indução definidas em 2.1 e 2.2. Quais são as diferenças essenciais entre as duas apresentações da regra de indução para o tipo Nat?* □

Abaixo segue a definição usual da função de adição por recursão primitiva. A indução nos construtores é feita no segundo argumento. Os rótulos `add01, add02` são opcionais, mas foram colocados para que pudéssemos fazer referência as regras (equações) durante a formalização das provas.

```
primrec add::"Nat ⇒ Nat ⇒ Nat"
  where
    add01:  "add x Z = x" |
    add02:  "add x (suc y) = suc (add x y)"
```

Note que a declaração do tipo da função indica que a função está na forma currificada, isto é, que os argumentos são passados um de cada vez. Ou seja, *add* aqui é uma função de alta ordem, isto é, *add* é uma função que recebe um argumento do tipo Nat e retorna uma função do tipo $Nat \Rightarrow Nat$. Em HOL, termos são formados como em programação funcional, isto é, aplicando-se funções em argumentos. Portanto se `f` é uma função do tipo $\sigma \Rightarrow \tau$ e x é to tipo σ, então o termo `f x` (note o espaço entre a função e o argumento) é do tipo τ. Por exemplo, `add (suc (suc Z))` é uma função que recebe um natural qualquer e soma dois a este valor. Desta forma, este termo tem tipo $Nat \Rightarrow Nat$.

Então com as declarações acima de *add* e Nat, teriamos as seguintes representações sintáticas para as segintes expressões matemáticas tradicionais:

Expressão	Termo
$1 + 0$	`add (suc Z) Z`
$1 + (2 + 1)$	`add (suc Z) (add (suc (suc Z)) (suc Z))`
$(1 + 1) + 1$	`add (add (suc Z) (suc Z)) (suc Z)`

Comentário 2.1. *Como é comum em lógica, Isabelle distingue variáveis livres e ligadas. Variáveis ligadas são automaticamente renomeadas para evitar conflitos com variáveis livres. Além disso, Isabelle possui um terceiro tipo de variável, chamada de* ***variável esquemática*** *or* ***desconhecida****, a qual deve sempre ter o ponto de interrogação* `?` *como seu primeiro caracter. Em termos lógicos, uma variável esquemática é uma variável livre. Entretanto, durante o processo de uma prova, ela pode ser instanciada por qualquer outro termo (do tipo correto).* □

O conjunto de regras de simplificação (reescrita) associadas a declaração acima e que são utilizadas por Isabelle é representada pelas equações abaixo.

```
add ?x Z = ?x
add ?x (suc ?y) = suc (add ?x ?y)
```

Note que as variáveis livres na definição acima, agora são interpretadas como variáveis esquemáticas por Isabelle.

Exercício 2.2. *Compute (manualmente) de acordo com as equações da função add logo acima, o resultado das seguintes expressões. Não esquecer de indicar as substituições e as equações utilizadas.*

1. `add Z (suc (suc Z))` *Observe a seguinte computação:*

$$\begin{array}{rcll} \texttt{add Z (suc (suc Z))} & = & \texttt{suc (add Z (suc Z))} & \textit{(by add02, } [x := Z, y := suc\ Z]\textit{)} \\ \ldots & = & \texttt{suc (suc (add Z Z))} & \textit{(by add02, } [x := Z, y := Z]\textit{)} \\ \ldots & = & \texttt{suc (suc Z)} & \textit{(by add01,} [x := Z]\textit{)} \end{array}$$

A computação acima é chamada de raciocínio equacional. Note que os três pontos (. . .) funcionam como uma abreviação para a expressão (termo) do lado direito da equação que está logo acima. Implicitamente, temos que o termo da esquerda da primeira equação é igual ao termo da direita da última equação através da aplicação implícita (por duas vezes) da regra de transitividade da igualdade.

2. `add (suc (suc Z)) (suc (suc Z))`

3. `add Z (add (suc (suc Z)) (suc Z))`

4. `add (add Z (suc (suc Z))) (suc Z)`

5. *Prove os teoremas abaixo informalmente utilizando a técnica de indução.*

$\forall x :: Nat. \forall y :: Nat. \forall z :: Nat.$ `add x (add y z) = add (add x y) z`	*Th-add-01*
$\forall x :: Nat. \forall y :: Nat.$ `add x y = add y x`	*Th-add-02*
$\forall x :: Nat.$ `add Z x = x`	*Th-add-03*
$\forall x :: Nat. \forall y :: Nat.$ `add x (suc y) = add (suc x) y`	*Th-add-04*
$\forall x :: Nat. \forall y :: Nat.$ `add (suc x) y = suc (add x y)`	*Th-add-05*

□

Solução 2.1 (Teorema `Th-add-01`)**.** *Agora faremos uma discussão detalhada do teoirema* ***Th-add-01*** *(associtividade da função add) para que você entenda os princípios que estão envolvidos. De acordo com a regra definida em 2.2, temos que colocar o teorema* ***Th-add-01*** *na forma da conclusão da regra* **Ind-Nat**. *Ou seja, temos que colocar o teorema* ***Th-add-01*** *no formato*

$$\forall n :: Nat.\ P\ n$$

Isto significa que devemos, especificar a propriedade acima na forma $P\ n$, onde n é uma variável livre (não está no escopo de nenhum quantificador) em P. Normalmente, n é a variável sobre a qual faremos a indução. Como regra geral, recomenda-se escolher que a variável de indução seja na mesma posição na qual a definição da função primitiva recursiva foi feita.

$$P\ n \stackrel{def}{=} \forall x :: Nat. \forall y :: Nat.\ \texttt{add x (add y n) = add (add x y) n}$$

onde a variável original ***z*** *(pela sua posição) foi escolhida como a variável de indução. Desta forma temos que*

$$\begin{array}{l} \forall n :: Nat.\ P\ n \\ \stackrel{def}{=} \forall n :: Nat. \forall x :: Nat. \forall y :: Nat.\ \texttt{add x (add y n) = add (add x y) n} \end{array}$$

Agora, voltando para a regra **Ind-Nat**, *vemos que para provar* $\forall n :: Nat.\ P\ n$, *precisamos provar dois sub-objetivos (subgoals):*

Caso	*Objetivo (Goal)*
Base	$P\ Z$
Indutivo	$(P\ x_0) \rightarrow P\ (suc\ x_0)$

Desta forma, dada uma propriedade P *qualquer, para prová-la por indução estrutural sobre os construtores de dados de* `Nat` *é preciso sempre fazer os seguintes passos:*

1. *Definir a propriedade que queremos provar em função da variável onder será feita a indução.*

$$P\ n \stackrel{def}{=} \forall x :: Nat. \forall y :: Nat.\ \texttt{add x (add y n) = add (add x y) n}$$

2. *Definir as propriedades relativas aos casos base, hipótese de indução e caso indutivo. Desta forma, temos:*

$$\begin{array}{lll} P\ Z & \stackrel{def}{=} & \forall x :: Nat. \forall y :: Nat.\ \texttt{add x (add y Z) = add (add x y) Z} \\ & & (\textit{Caso Base}) \\ P\ x_0 & \stackrel{def}{=} & \forall x :: Nat. \forall y :: Nat.\ \texttt{add x (add y } x_0\texttt{) = add (add x y) } x_0 \\ & & (\textit{Hipótese de Indução}) \\ P\ (suc\ x_0) & \stackrel{def}{=} & \forall x :: Nat. \forall y :: Nat.\ \texttt{add x (add y (suc } x_0\texttt{))} \\ & & = \texttt{add (add x y) (suc } x_0\texttt{)} \\ & & (\textit{Caso Indutivo}) \end{array}$$

3. *Prova* $P\ Z$ *. Sejam* $x_0 :: Nat, y_0 :: Nat$ *valores fixos mas arbitrários. Então é suficiente mostrar que* $\texttt{add } x_0\ \texttt{(add } y_0\ \texttt{Z) = add (add } x_0\ y_0\texttt{) Z}$. *Agora, observe que:*

$$\begin{array}{rcll} \texttt{add } x_0\ \texttt{(add } y_0\ \texttt{Z)} & = & \texttt{add } x_0\ y_0 & \textit{(by add01, } x := y_0\textit{)} \\ & = & \texttt{add (add } x_0\ y_0\texttt{) Z} & \textit{(by add01, } x := \texttt{add } x_0\ y_0\textit{)} \end{array}$$

q.e.d

4. *Prova* $(P\ x_0) \rightarrow P\ (suc\ x_0)$.

   Sejam $x0 :: Nat, x_1 :: Nat, y_1 :: Nat$ *valores fixos mas arbitrários. Então é suficiente mostrar que* $add\ x_1\ (add\ y_1\ (suc\ x_0)) = add\ (add\ x_1\ y_1)\ (suc\ x_0)$. *Agora, observe que:*

$$\begin{array}{rcl} \texttt{add } x_1\ \texttt{(add } y_1\ \texttt{(suc } x_0\texttt{))} & = & \texttt{add } x_1\ \texttt{(suc (add } y_1\ x_0\texttt{))} \\ & & \textit{(by add02, } x := y_1, y := x_0\textit{)} \\ & = & \texttt{suc (add } x_1\ \texttt{(add } y_1\ x_0\texttt{))} \\ & & \textit{(by add02, } x := x_1, y := \texttt{add } y_1\ x_0\textit{)} \\ & = & \texttt{suc (add (add } x_1\ y_1\texttt{) } x_0\texttt{)} \\ & & \textit{(by IH,}\forall E, x := x_1, y := y_1\textit{)} \\ & = & \texttt{add (add } x_1\ y_1\texttt{) (suc } x_0\texttt{)} \\ & & \textit{(by add02, } x := \texttt{add } x_1\ y_1, y := x_0\textit{)} \end{array}$$

q.e.d

Comentário 2.2. *Observe que as variáveis arbitrárias* $x_1 :: Nat, y_1 :: Nat$ *mencionadas acima, durante a prova de* $P\ (suc\ x_0)$*, referem-se a aplicação da regra de introdução do quantificador universal (*$\forall I$*) por duas vezes, como mostramos logo abaixo:*

	$x_1 :: Nat$	
	$y_1 :: Nat$	
	$\vdots$	
	$\vdots$	
	$add\ x_1\ (add\ y_1\ (suc\ x_0)) = add\ (add\ x_1\ y_1)\ (suc\ x_0)$	
	$\forall y :: Nat.\ add\ x_1\ (add\ y\ (suc\ x_0)) = add\ (add\ x_1\ y)\ (suc\ x_0)$	$\forall$ *intro*
	$\forall x :: Nat. \forall y :: Nat.\ add\ x\ (add\ y\ (suc\ x_0)) = add\ (add\ x\ y)\ (suc\ x_0)$	$\forall$ *intro*

Note que a variável $x_0 :: Nat$ *já fora introduzida pela aplicação da mesma regra, graças a regra de indução para* Nat*. Uma observação similar vale para a prova de* $P\ Z$*.* □

Comentário 2.3. *As provas acima, utilizando o raciocínio equacional são de certa forma informais, porque as regras de introdução do quantificador universal são mencionadas com uso de linguagem natural e porque o raciocínio equacional utilizado acima, abstrai (ou deixa implícito) uma série de detalhes com relação a utilização das regras equacionais. Por exemplo, uma prova formal de* $P\ Z$*, no cálculo de dedução natural, deveria ser apresentada como segue:*

1	$x_0 :: Nat$	
2	$y_0 :: Nat$	
3	$add\ y_0\ Z = y_0$	*by add01,*$x := y_0$
4	$add\ x_0\ (add\ y_0\ Z) = add\ x_0\ (add\ y_0\ Z)$	*by =intro*
5	$add\ x_0\ (add\ y_0\ Z) = add\ x_0\ y_0$	*by 3,4,=elim*
6	$add\ (add\ x_0\ y_0)\ Z = add\ x_0\ y_0$	*by add01,*$x := add\ x_0\ y_0$
7	$add\ (add\ x_0\ y_0)\ Z = add\ (add\ x_0\ y_0)\ Z$	*by =intro*
8	$add\ x_0\ y_0 = add\ (add\ x_0\ y_0)\ Z$	*by 6,7,=elim*
9	$add\ x_0\ (add\ y_0\ Z) = add\ (add\ x_0\ y_0)\ Z$	*by 5,8,=trans*
10	$\forall y :: Nat.\ add\ x_0\ (add\ y\ Z) = add\ (add\ x_1\ y)\ Z$	*by 9,* $\forall$ *intro*
11	$\forall x :: Nat. \forall y :: Nat.\ add\ x\ (add\ y\ Z) = add\ (add\ x\ y)\ Z$	*by 10,* $\forall$ *intro*

□

2.2 Verificação com scripts de comandos

O objetivo desta seção é utilizar a demonstração do teorema da associatividade para introduzirmos o primeiro estilo de formalização de provas em Isabelle, isto é, utilizando scripts (de

baixo nível) que consistem em comandos e táticas para manipulação de estados de prova (proof states).

theorem `th_add01as : "∀ x y. add (add x y) z = add x (add y z)"`

O comando **theorem** acima faz essencialmente duas coisas:

- Ele estabelece um novo teorema a ser provado, isto é,

  `∀ x y. add (add x y) z = add x (add y z)`

- Note que quantificações aninhadas podem ser abreviadas. Isto é, fórmulas do tipo `∀x. ∀y. ∀z. P` podem ser abreviadas como `∀x y z. P`.
- Ele dá a esse teorema um nome, *th_add01as*, para referência futura (o sufixo `as` significa `apply-style`).

A reação de Isabelle ao comando acima é imprimir o estado inicial da prova, o qual consiste de alguma informação de cabeçalho (tipicamente quantos subojetivos devem ser resolvidos) seguido de:

```
1. ∀x y. add (add x y) z = add x (add y z)
```

Até que tenhamos terminado a prova, o estado da prova sempre se parece com a seguinte estrutura:

$$\begin{array}{ll} 1. & G_1 \\ \vdots & \\ n & G_n \end{array}$$

As linhas numeradas contem os subjetivos $G_1, \ldots, G_n$ que precisamos provar a fim de estabelecer o objetivo principal. Inicialmente, existe apenas um subobjetivo o qual é idêntico ao objetivo principal.

Voltando agora ao nosso objetivo principal, sabemos que propriedades de função definidas recursivamente são estabelecidas, isto é, provadas por indução. Em nosso caso, de forma a capitalizar sobre a definição por casos no segundo argumento da função *add*, faremos a indução óbvia sobre a variável z.

apply `(induct z)`

O comando acima diz a Isabelle para realizar a indução sobre a variável z. O sufixo `tac` significa `tática`, um sinômimo para uma função para prova de teoremas. Por default, indução atua sobre o primeiro subobjetivo. Após a execução deste comando, o novo estado da prova contém dois novos subobjetivos, um para o caso base (Z) e outro para o passo de indução ($Cons$).

```
1. ∀x y. add (add x y) Z = add x (add y Z)
2. ⋀z. ∀x y. add (add x y) z = add x (add y z) ⟹
       ∀x y. add (add x y) (suc z) = add x (add y (suc z))
```

O passo de indução é um exemplo de um formato geral de um subobjetivo:

$$i. \quad \bigwedge x_1 \ldots x_n.\ \mathit{premissas} \Longrightarrow \mathit{conclusão}$$

O prefixo de variáveis declaradas pelo quantificador $\bigwedge$ é tratado como uma lista de variáveis livres que são locais a este subobjetivo. Estas variávais locais são tratadas como **arbitrárias**, isto é, com valor desconhecido. Desta forma, uma declaração do tipo $\bigwedge$ `x::`τ em um determinado contexto de uma prova expressa exatamente que x é uma variável arbitrária do tipo τ. As *premissas* são as hipóteses locais para este objetivo e a *conclusão* é a proposição real a ser provada. No caso acima, a variável local (cujo nome é escolhido por Isabelle) é Nat e a premissa é a hipótese de indução. Caso haja

múltiplas premissas, elas são delimitadas pelos parênteses semânticos ⟦ e ⟧ e separadas por ponto e vírgula.
Agora, de forma a resolvermos o primeiro objetivo, acionamos o simplificador com o seguinte comando:

apply `(simp)` — resolve o caso base

Agora o estado da prova apresenta apenas mais um objetivo, como segue:

```
1. ⋀z. ∀x y. add (add x y) z = add x (add y z) ⟹
       ∀x y. add (add x y) (suc z) = add x (add y (suc z))
```

Chamando mais uma vez o simplificador, resolvemos o caso de indução,

apply `(simp)` — resolve o caso indutivo

```
No subgoals!
```

como mostra o estado da prova, isto é, não há mais subobjetivos e a prova pode ser finalizada como o comando **done**.

— lista de subobjetivos vazia: prova completa
done

Note que após a execução do comando **done**, Isabelle registra o teorema acima, de tal forma que a variável livre z (onde foi feita a indução), agora torna-se uma variável esquemática.

```
∀(x::Nat) y::Nat. add (add x y) (?z::Nat) = add x (add y ?z)
```

2.3 Verificação com a linguagem Isar

A prova interativa de teoremas foi sempre dominada por um modelo de prova que se originou no sistema LCF [1], i.e., uma prova é uma seqüência mais ou menos estruturada de comandos que manipulam um estado de prova (e.g., como na prova acima). Portanto, o texto da prova é adequado somente para uma máquina. Para um ser humano, a prova só ganha vida quando ele (ou ela) pode ver as alterações de estado causadas pelo execução passo a passo dos comandos. Por isso que neste tutorial, documentamos a prova acima com frequentes inspeções ao estado da prova. Este estilo de prova pode ser comparado com programas em linguagem assembly. Em Isabelle, como vimos anteriormente, a encarnação deste estilo é uma seqüência de comandos **apply**.
Por outro lado, existe o modelo de linguagem de provas semelhantes a que fazemos em matemática, como desenvolvido na solução 2.1, cuja mecanização foi explorada inicialmente pelo sistema Mizar [4] e depois por Isar[1] [8]. Os argumentos mais importantes para este estilo são a comunicação e manutenção. Provas estruturadas são imensamente mais legíveis e passíveis de manutenção do que scripts **apply**.
No que segue assumimos familiariade com dedução natural, especialmente regras de introdução e eliminação do quantificador universal, do conceito de substituição e especialmente

[1] Isar significa `Intelligible semi-automated reasoning`

de provas com raciocínio sobre equações (calculational reasoning), isto é, uma sequência de equações onde as aplicações da regra de transitividade estão implícitas.

A linguagem Isar é extremamente rica e complexa. Aqui estaremos utilizando apenas um pequeno fragmento da linguagem, e portanto é suficiente explicarmos a linguagem diretamente através de exemplos. Para uma visão mais ampla da expressividade de Isar, sugerimos [2, 5] e especialmente [7].

Abaixo vemos uma gramática simplificada para provas em Isar. Parênteses são utilizados para agrupamento e `?` para indicar um ítem opcional.

$$
\begin{array}{lll}
\textit{proof} & ::= & \textbf{proof}\ \ \textit{method}?\ \ \textit{statement}^*\ \ \textbf{qed} \\
\textit{statement} & ::= & \textbf{fix}\ \ \textit{variables} \\
 & & \mid \textbf{assume}\ \ \textit{propositions} \\
 & & \mid (\textbf{from}\ \ \textbf{facts})?\ \ (\textbf{show} \mid \textbf{have})\ \ \textit{propositions}\ \ \textit{proof} \\
\textit{proposition} & ::= & (\textit{label}\ :)?\ \ \textit{string} \\
\textit{fact} & ::= & \textit{label} \\
\textit{method} & ::= & - \mid \textbf{this} \mid \textbf{auto} \mid \textbf{simp} \mid \textbf{blast} \mid \textbf{simp\_all} \\
 & & \mid \textbf{rule}\ \textit{fact} \mid \textbf{induct}\ \textit{variable} \mid \ldots
\end{array}
$$

Uma prova pode ser tanto composta (**proof-qed**) ou atômica (**by**). Um template típico de uma prova é como abaixo:

proof
  assume ‘‘premissas’’
  have ‘‘...’’
  ⋮
  have ‘‘...’’
  show ‘‘conclusão’’
qed

O template acima prova a conjectura premissas $\Longrightarrow$ conclusão. Os comandos intermediários **have** estão ali para preencher (com resultados intermediários) o hiato entre as premissas e a conclusão. Por outro lado, **show** estabelece a conclusão do teorema

Considere agora a prova do teorema anterior, no mesmo nível de detalhe, mas agora utilizando a linguagem Isar.

```
theorem th_add01isA : "∀ x y.  add (add x y) z = add x (add y z)"
  proof(induct z)
```

Observe que agora temos o seguinte estado de prova:

```
1. ∀x y. add (add x y) Z = add x (add y Z)
2. ⋀z. ∀x y. add (add x y) z = add x (add y z) ⟹
       ∀x y. add (add x y) (suc z) = add x (add y (suc z))
```

— prova do caso base
```
  show "∀ x y. add (add x y) Z = add x (add y Z)" by simp
next
```

O comando **next** sempre seleciona o próximo subobjetivo que está na fila. Por exemplo, note agora que o próximo objetivo é o passo de indução:

```
1. ⋀z. ∀x y. add (add x y) z = add x (add y z) ⟹
       ∀x y. add (add x y) (suc z) = add x (add y (suc z))
```

Como z deve ser arbitrário, fixamos uma variável nova, no caso x_0

```
  fix x0::Nat — elemento arbitrário, mas fixo
  assume IH: "∀ x y. add (add x y) x0 = add x (add y x0)"
  show "∀ x y. add (add x y) (suc x0) = add x (add y (suc x0))"
    by (simp add:IH)
qed
```

Veja que agora temos uma prova em um nível de abstração independente da máquina e que reflete a notação matemática usual para este tipo de demonstração. Note que na prova (atômica) do passo indutivo, utilizamos um modificador no método de simplificação. Além das regras `add01,add02` utilizadas por default, adicionamos ainda a hipótese de indução.

O próximo passo agora e construírmos uma prova em Isar onde possamos registrar passos atômicos de raciocínio, isto é, onde cada dedução de uma nova equação seja conseqüência da aplicação de uma única regra de simplificação. A idéia essencial aqui é que queremos utilizar Isar com uma verificador das nossas provas (proof checker).

Portanto, aqui segue o teorema da associatividade em detalhe.

```
theorem th_add01isB:"∀ x y.  add (add x y) z = add x (add y z)"
  proof (induct z)
    show "∀ x y. add (add x y) Z = add x (add y Z)"
```

Agora a prova do caso base não pode ser atômica. Tempos que aplicar duas vezes a regra de introdução do quantificador universal, o que implica que a prova deve ser feita para quaisquer dois elementos arbitrários, mas fixos.

```
    proof (rule allI, rule allI)
      fix x0::Nat and y0::Nat
      have "add (add x0 y0) Z = add x0 y0" by (simp only:add01)
      also have "...  = add x0 (add y0 Z)" by (simp only:add01)
```

Duas observações importantes cabem aqui: primeiramente, observe que ao invocarmos o método de simplificação, observe que permitimos somente o uso da primeira equação da definição de *add*. Na linha exatamente acima, a expressão (três pontos) "..." serve para fazer referência ao termo do lado direito da equação que está imediatamente acima, poupando a nós, pelo menos um pouco de digitação. Além disso o comando **also have** possui duas funções: primeiramente a de estabelecer a proposição via **have**. O comando **also** tem a função de estabelecer implicitamente o fecho transitivo entre a equação estabelecida por **have** da própria linha e o comando **have** da linha anterior. Desta forma o que está deduzido na linha acima é que `add (add x0 y0) Z = add x0 (add y0 Z)`

```
      finally show "add (add x0 y0) Z = add x0 (add y0 Z)" by simp
```

Finalmente, o comando acima tem a função de registrar explicitamente o objetivo deste contexto da prova que já estado implicitamente calculado na linha anterior. É essa proposição que é exportada como a conclusão deste contexto (bloco, caixa) da prova. O resto da prova agora se desenvolve de forma similar.

```
  qed
next — pega o próximo subojetivo, isto é, o passo de indução
  fix z0::Nat
  assume IH: "∀ x y. add (add x y) z0 = add x (add y z0)"
  show "∀ x y. add (add x y) (suc z0) = add x (add y (suc z0))"
```

```
    proof (rule allI, rule allI)
      fix x0::Nat and y0::Nat
      have "add (add x0 y0) (suc z0) = suc (add (add x0 y0) z0)"
       by (simp only:add02)
      also have "...  = suc (add x0 (add y0 z0))" by (simp only:IH)
      also have "... = add x0 (suc (add y0 z0))" by (simp only:add02)
      also have "... = add x0 (add y0 (suc z0))" by (simp only:add02)
      finally
      show "add (add x0 y0) (suc z0) = add x0 (add y0 (suc z0))" by simp
  qed
qed
```

A seguinte definição recursiva tem por objetivo computar a multiplicação de dois termos do tipo Nat:

```
primrec mult::"Nat ⇒ Nat ⇒ Nat" where
  mult01: "mult x Z = Z" |
  mult02: "mult x (suc y) = add x (mult x y)"
```

Exercício 2.3. *Com base na definição da função de multiplicação, compute o valor das seguintes expressões:*

1. `mult Z (suc (suc Z))`

2. `mult (suc (suc Z)) (suc (suc (suc Z)))`

□

Exercício 2.4. *Com base na definição de multiplicação, prove por indução os seguintes teoremas:*

$\forall x :: Nat.\ \forall y :: Nat.\ \forall z :: Nat.\ mult\ x\ (mult\ y\ z) = mult\ (mult\ x\ y)\ z$	*Th-mult-01*
$\forall x :: Nat.\ \forall y :: Nat.\ mult\ x\ y = mult\ y\ x$	*Th-mult-02*
$\forall x :: Nat.\ \forall y :: Nat.\ \forall z :: Nat.\ mutl\ x\ (add\ y\ z) = add\ (mult\ x\ y)\ (mult\ x\ z)$	*Th-mult-03*

□

2.4 Pit Stop

Até esta parte do tutorial, procuramos apresentar os requisitos essenciais de Isabelle para que o leitor possa construir por si mesmo provas por indução estrutural sobre funções recursivas construidas por tipos de dados indutivos. O estilo script é muito favorável para os estágios iniciais de exploração da possível validade de uma conjectura lógica. Este estilo também é muito útil quando queremos explorar os possíveis lemas que eventualmente podem ser necessários para a prova de um determinado teorema. Por outra lado, com Isar, o usuário pode construir documentos de alto nível, que além de serem extremamente legíveis, admitem a imposição de estrutura, isto é, da documentação de **subprovas**. Mais do que isso, devido a riqueza e a expressividade da linguagem Isar, é possível construir demonstrações em grande

detalhe, de forma que provas neste nível de abstração podem servir para verificarmos nossos raciocínios passo a passo, normalmente expressos em provas construídas com lápis e papel.

No que segue, estaremos prosseguindo com as mesmas ferramentas, mas agora trilhando outros caminhos, isto é, trabalhando sobre listas e árvores genéricas.

end

3 Uma Teoria para Listas Genéricas

theory `list_tutorial`
imports `nat_tutorial`
begin

3.1 Definições Básicas

Aqui procedemos como feito anteriormente, apresentamos a definição indutiva da nossa versão de listas genéricas, e logo após a definição recursiva de uma simples função sobre listas (no caso, concatenação).

```
datatype 'a  List  = Empty | cons 'a "'a List"
```

Na definição acima, *List* é o construtor do tipo e `'a` é um parâmetro, que como em ML [3], representa uma variável de tipo. Isto significa que estamos definindo listas polimórficas, ou genéricas. Os construtores de dados são, portanto, *Empty* e *cons*, onde o primeiro é um construtor nulário e o segundo, um construtor binário, que recebe um elemento do tipo , uma lista do mesmo tipo, e retorna um uma nova lista deste tipo, onde o novo elemento é adicionado à esquerda da lista. Por exemplo, podemos ter os seguintes exemplos de termos, onde *nat* é o tipo dos números naturais que é predefinido em Isabelle.

Tipo	Elementos do Tipo
`Nat List`	`Empty, cons Z Empty, cons (suc Z) Empty,...`
`nat List`	`Empty, cons 1 Empty, cons 10 (cons 4 Empty),...`
`(nat*Nat) List`	`Empty, cons (2, suc Z) Empty, cons (0, Z) Empty,...`
⋮	⋮

Definição 3.1. *A seguinte regra de indução é automaticamente associada ao tipo Nat, após o Isabelle ter processado a definição do tipo via o comando* ***datatype****.*

$$\frac{\texttt{P Empty} \qquad \bigwedge \texttt{e}_0\ \texttt{l}_0.\ \dfrac{\texttt{P l}_0}{\texttt{P (cons e}_0\ \texttt{l}_0\texttt{)}}}{\texttt{P l}}\ \mathsf{IndList}$$

□

Desta forma, lendo a regra IndList de baixo para cima, temos que, para provarmos uma propriedade P para qualquer elemento do tipo `'a` *List*, precisamos fazer apenas duas coisas:

1. Provar que P vale para *Empty*;

2. Assumindo um elemento arbitrário e_0 e que P vale para uma lista arbitrária l_0, provar que P também vale para `cons (e`$_0$`::'a) (l`$_0$`::'a List)`.

Definição 3.2. *Considere a regra para prova de indução do tipo de dado* `'a List`*, expressa no estilo de dedução natural sem o quantificador da meta-lógica* $\bigwedge$ *de Isabelle, onde* e_0 *e* l_0 *são variáveis arbitrárias que não podem ocorrer* ***livres*** *em nenhuma hipótese utilizada para provar* P $(cons\ e_0\ l_0)$*.*

$$\dfrac{\begin{array}{ccc} \vdots & \boxed{\begin{array}{c} e_0 :: \texttt{'a},\ l_0 :: \texttt{'a List},\ P\ l_0 \\ \vdots \\ P\ (cons\ e_0\ l_0) \end{array}} & (e_0, l_0\ local) \\ P\ Empty & & \end{array}}{\forall l :: \texttt{'a List}.\ P\ l}\ \textit{Ind-List}$$

□

Exercício 3.1. *Compare a regra acima com a regra de indução que Isabelle associa automaticamente à declaração do tipo de dado (datatype)* `'a List` *logo acima. Quais são as diferenças essenciais entre as duas apresentações da regra de indução para este tipo?* □

Abaixo, a definição recursiva da função de concatenção. A indução por casos é feita no primeiro argumento.

```
primrec cat::"'a List⇒'a List ⇒'a List"
  where
    cat01: "cat Empty l = l"|
    cat02: "cat (cons h t) l= cons h (cat t l)"
```

Em matemática discreta e em várias linguagens de programação funcional, listas são normalmente representadas como uma lista de lementos separadas por vírgulas e delimitadas por colchetes, enquanto a operação de concatenação é com freqüência denotada pelo operador `@` (`at`). Assumindo esta notação, segue abaixo uma tabela de como essa representação seria traduzida para a nossa sintaxe acima.

Expressão	Termo
`[ ]`	`Empty`
`[1,2,3]`	`cons 1 (cons 2 (cons 3 Empty))`
`[ ]@[1+2,0]`	`cat Empty (cons (add (suc Z) (suc (suc Z))) (cons Z Empty))`
`[1,2]@[3,4]`	`cat (cons 1 (cons 2 Empty)) (cons 3 (cons 4 Empty))`

Exercício 3.2. *Compute (manualmente) de acordo com as equações da função cat logo acima, o resultado das seguintes expressões. Não esquecer de indicar as substituições e as equações utilizadas.*

1. `cat (cons Z (cons Z Empty)) Empty`

$$\begin{aligned}
\texttt{cat (cons Z (cons Z Empty)) Empty} &= \texttt{cons Z (cat (cons Z Empty) Empty)} \\
&\quad (\textit{by cat02}, h := Z, t := \textit{cons } Z \textit{ Empty}, l := \textit{Empty}) \\
&\ldots = \texttt{cons Z (cons Z (cat Empty Empty))} \\
&\quad (\textit{by cat02}, h := Z, t := \textit{Empty}, l := \textit{Empty}) \\
&\ldots = \texttt{cons Z (cons Z Empty)} \\
&\quad (\textit{by cat01}, l := \textit{Empty})
\end{aligned}$$

2. `cat (cons 1 (cons 2 Empty)) (cons 3 (cons 4 Empty))`

3. `cat (cons 1 Empty) Empty`

□

Exercício 3.3. *Prove os teoremas abaixo informalmente utilizando a técnica de indução estrutural. Note que as quantificações aninhadas estão abreviadas, isto é,* `∀x y z. P` *significa* $\forall x.\ \forall y.\ \forall z.\ P$.

`∀l1 l2 l3. cat l1 (cat l2 l3) = cat (cat l1 l2) l3`	*Th-cat-01*
`∀l1 l2. len (cat l1 l2) = add (len l1) (len l2)`	*Th-cat-02*
`∀l. cat l Empty = l`	*Th-cat-03*
`∀l1 l2. natLen (cat l1 l2) = natLen l1 + natLen l2`	*Th-cat-04*
`∀l. reverse (reverse l) = l`	*Th-rev-01*
`∀l1 l2. reverse (cat l1 l2) = cat (reverse l2) (reverse l1)`	*Th-rev-02*

onde

- *len* :: *'a List* ⇒ *Nat* *é uma função que recebe uma lista e retorna a quantidade de elementos desta lista;*
- *natLen* :: *'a List* ⇒ *nat* *é uma função que recebe uma lista e retorna a quantidade de elementos desta lista utilizando os naturais fornecidos por Isabelle.*
- *reverse* :: *'a List* ⇒ *'a List* *é uma função que recebe uma lista e retorna a lista com os elementos em ordem reversa.*

□

Solução 3.1 (Teorema `Th-cat-01`)**.** *Primeiramente, definimos a propriedade em função da variável de indução e após instanciamos de acordo com os construtores do tipo.*

1. $P\ l \stackrel{def}{=} \forall$`l2 l3. cat l (cat l2 l3) = cat (cat l l2) l3`
2. $P\ \textit{Empty} \stackrel{def}{=} \forall$`l2 l3. cat Empty (cat l2 l3) = cat (cat Empty l2) l3`
3. $P\ l0 \stackrel{def}{=} \forall$`l2 l3. cat l0 (cat l2 l3) = cat (cat l0 l2) l3`
4. $P\ (\textit{cons}\ e0\ l0) \stackrel{def}{=} \forall$`l2 l3. cat (cons e0 l0) (cat l2 l3) = cat (cat (cons e0 l0) l2) l3`

5. *Prova P Empty: ...* `q.e.d`

6. *Prova* $(P\ l0) \rightarrow P(cons\ e0\ l0)$*: ...* `q.e.d`

□

3.2 Verificação com scripts de comandos

Abaixo, provamos o teorema da associtividade da operação de concatenação utilizando a linguagem de comandos do Isabelle. As explicações são análogas as que escrevemos na prova sobre a associtividade da adição. A única diferença é que abaixo utilizamos o método **simp_all** para simplicar todos (dois) subobjetivos de uma só vez.

```
theorem th_cat01as: "∀ l2 l3. cat l (cat l2 l3) = cat (cat l l2) l3"
   apply (induct l)
   apply (simp_all)
done
```

3.3 Verificação com a linguagem Isar

Abaixo, agora utilizando a linguagem isar para provar novamente o teorema da associatividade em duas versões. Na primeira, no mesmo nível de abstração da prova anterior. E na segunda, em detalhes, utilizando passos atômicos de raciocínio (simplificação), como feito com lápis e papel.

```
theorem th_cat01isA: "∀ l2 l3. cat l (cat l2 l3) = cat (cat l l2) l3"
   proof (induct l)
     show "∀ l2 l3. cat Empty (cat l2 l3) = cat (cat Empty l2) l3" by simp
   next
     fix e0 and l0
```

Note que acima não anotamos os tipos das várias $e0$ e $l0$, já que Isabelle pode inferí-los automaticamente para nós.

```
     assume IH: "∀ l2 l3. cat l0 (cat l2 l3) = cat (cat l0 l2) l3"
     show "∀ l2 l3. cat (cons e0 l0) (cat l2 l3) = cat (cat (cons e0 l0) l2) l3"
           by (simp add:IH)
   qed

 theorem th_cat01isB: "∀ l2 l3. cat l (cat l2 l3) = cat (cat l l2) l3"
   proof (induct l)
     show "∀ l2 l3. cat Empty (cat l2 l3) = cat (cat Empty l2) l3"
       proof (rule allI, rule allI)
         fix k and  m
```

Desta vez, ao contrário da prova de associatividade da adição, durante a computação da cadeia de equações, não estaremos utilizando a abreviação "..." para fazer referência ao termo direito da equação anterior. Digitaremos a expressão explicitamente.

```
         have "cat Empty (cat k m) = cat k m" by (simp only:cat01)
         also  have "cat k m = cat (cat Empty k) m" by (simp only:cat01)
         finally show "cat Empty (cat k m) = cat (cat Empty k) m" by this
       qed
```

```
  next
    fix e0::'a and l0::"'a List"
    assume IH: "∀ l2 l3. cat l0 (cat l2 l3) = cat (cat l0 l2) l3"
    show "∀ l2 l3. cat (cons e0 l0) (cat l2 l3) = cat (cat (cons e0 l0) l2) l3"
      proof (rule allI, rule allI)
        fix k and m
        have "cat (cons e0 l0) (cat k m) = cons e0 (cat l0  (cat k m))"
             by (simp only:cat02)
        also have "cons e0 (cat l0 (cat k m)) = cons e0 (cat (cat l0 k) m)"
             by (simp only:IH)
        also have "cons e0 (cat (cat l0 k) m) = cat (cons e0 (cat l0 k)) m"
             by (simp only:cat02)
        also  have "cat (cons e0 (cat l0 k)) m = cat (cat (cons e0 l0) k) m"
             by (simp only:cat02)
        finally  show "cat (cons e0 l0) (cat k m) =cat (cat (cons e0 l0) k) m"
             by this
     qed
qed
```

3.4 Pit Stop

Nesta parte do tutorial, utilizamos a mesma metodologia já discutida em detalhes na teoria sobre naturais para desenvolver uma pequena teoria sobre listas, com seus respectivas declarações para tipos, funções e teoremas. Quanto ao último, utilizamos novamente a linguagem de comandos e a linguagem Isar para explorar e construir as provas. Adiante com as árvores binárias!

end

4 Uma Teoria para Árvores Genéricas

```
theory tree_tutorial
imports list_tutorial
begin
```

4.1 Definições Básicas

Aqui procedemos como feito anteriormente, apresentamos a definição indutiva da nossa versão de listas genéricas, e logo após a definição recursiva de uma simples função sobre árvores (no caso, a função de espelhamento ou reflexão).

```
datatype 'a Tree = Leaf | Br  'a "'a Tree" "'a Tree"
```

Na definição acima, *Tree* é o construtor do tipo e `'a` é um parâmetro, que representa uma variável de tipo. Isto significa que estamos definindo árvores binárias polimórficas, ou genéricas. Os construtores de dados são, portanto, *Leaf* e *Br* (de `Branch`), onde o primeiro é um construtor nulário que é do tipo arvore e o segundo, um construtor com três argumentos, que recebe um elemento do tipo , duas árvores binárias do mesmo tipo retornando uma nova árvore, com o elemento do tipo sendo a nova raiz desta árvore.

Definição 4.1. *A regra de indução associada a declaração do* ***datatype*** *`'a Tree` acima é da seguinte forma:*

$$\frac{\texttt{P Leaf} \qquad \bigwedge \texttt{x}_0\ \texttt{lt}_0\ \texttt{rt}_0.\ \dfrac{\texttt{P lt}_0 \qquad \texttt{P rt}_0}{\texttt{P (Br x}_0\ \texttt{lt}_0\ \texttt{rt}_0\texttt{)}}}{\texttt{P t}}\ \textsf{IndTree}$$

□

Desta forma, lendo a regra IndTree de baixo para cima, temos que, para provarmos uma propriedade P para qualquer árvore t do tipo `'a Tree`, precisamos fazer apenas duas coisas:

1. Provar que P vale para `Leaf`;
2. Assumindo um elemento arbitrário x_0 e que P vale para aŕvores binárias arbitrárias lt_0 e rt_0, provar que P também vale para `Br x`$_0$ `lt`$_0$ `rt`$_0$.

A mesma regra apresentada acima, pode ser representada na notação de dedução natural sem o quantificador da meta-lógica $\bigwedge$ da seguinte forma:

Definição 4.2.

$$\frac{\begin{array}{ccc} \vdots & \boxed{\begin{array}{c} x_0 :: \texttt{'a},\ lt_0, rt_0 :: \texttt{'a Tree}, P\ rt_0, P\ lt_0 \\ \vdots \\ P\ (Br\ x_0\ lt_0\ rt_0) \end{array}} & \\ P\ Leaf & & (x_0, lt_0, rt_0\ local) \end{array}}{\forall t :: \texttt{'a Tree}.\ P\ t}\ \textit{Ind-Tree}$$

□

Abaixo, a definição primitiva recursiva da função de reflexão.

```
primrec reflect::"'a Tree ⇒ 'a Tree"
   where
     ref01:"reflect Leaf = Leaf" |
     ref02:"reflect (Br label lt rt) = Br label (reflect rt) (reflect lt)"
```

Em matemática discreta árvores são normalmente representadas como uma lista de elementos separadas por vírgulas e delimitadas por parênteses angulares. Assumindo esta notação, segue abaixo uma tabela de como essa representação seria traduzida para a nossa sintaxe acima.

Expressão	Termo
$\langle\rangle$	`Leaf`
$\langle 1, \langle\rangle, \langle\rangle\rangle$	`Br 1 Leaf Leaf`
$\langle 1, \langle 2, \langle\rangle, \langle\rangle\rangle, \langle 3, \langle\rangle, \langle\rangle\rangle\rangle$	`Br 1 (Br 2 Leaf Leaf) (Br 3 Leaf Leaf)`

Exercício 4.1. *Prove os teoremas abaixo informalmente (com lápis e papel) utilizando a técnica de indução estrutural. Note que as quantificações aninhadas estão abreviadas, isto é,* `∀x y z. P` *significa* $\forall x.\ \forall y.\ \forall z.\ P$.

∀t. reflect (reflect t) = t	*Th-refl-01*
∀t. nodesNat (reflect t) = nodesNat t	*Th-nodes-01*
∀t. leavesNat t = suc (nodesNat t)	*Th-nodes-02*
∀t. numNodes (reflect t) = numNodes t	*Th-nodes-03*
∀t. numLeaves t = numNodes t + 1	*Th-nodes-04*

onde

- *nodesNat* :: *'a Tree* ⇒ *Nat* *é uma função que recebe uma árvore e retorna o número de nodos desta árvore;*
- *leavesNat* :: *'a Tree* ⇒ *Nat* *é uma função que recebe uma árvore e retorna o número de folhas desta árvore;*
- *numNodes* :: *'a Tree* ⇒ *nat* *é uma função que recebe uma árvore e retorna o número de nodos desta árvore utilizando os naturais definidos por Isabelle;*
- *numLeaves* :: *'a Tree* ⇒ *nat* *é uma função que recebe uma árvore e retorna o número de folhas desta árvore utilizando os naturais definidos por Isabelle;*

□

Solução 4.1 (Teorema `Th-refl-01`)**.** *Primeiramente, definimos a propriedade em função da variável de indução e após instanciamos de acordo com os construtores do tipo.*

1. $P\ t \stackrel{def}{=}$ `reflect (reflect t) = t`
2. $P\ Leaf \stackrel{def}{=}$ `reflect (reflect Leaf) = Leaf`
3. $P\ lt0 \stackrel{def}{=}$ `reflect (reflect lt0) = lt0`
4. $P\ rt0 \stackrel{def}{=}$ `reflect (reflect rt0) = rt0`
5. $P\ (Br\ x0\ lt0\ rt0) \stackrel{def}{=}$ `reflect (reflect (Br x0 lt0 rt0)) = Br x0 lt0 rt0`
6. *Prova P Leaf: . . .* *q.e.d*
7. *Prova* $(P\ lt0) \wedge (P\ rt0) \rightarrow P(Br\ x0\ lt0\ rt0)$*: . . .* *q.e.d*

□

4.2 Verificação com scripts de comandos

Abaixo declaramos o teorema a ser provado. Como nos outros casos, a prova é imediata. Desta vez, utilizamos o método **auto** ao invés de **simp**. O primeiro pode ser visto como uma versão sofisticada do segundo. Essencialmente, **auto** tenta resolver tantos subobjetivos quanto for possível, utilizando os métodos do simplificador além de raciocínio lógico elementar.

```
theorem th_refl01as: "reflect (reflect t) = t"
   apply (induct t)
   apply (auto)
done
```

4.3 Verificação com a linguagem Isar

No que segue, utilizando sempre a mesma metodologia, provamos o teorema acima nas duas versões já conhecidas, agora utilizando a linguagem Isar. Observe, que abaixo precisamos necessariamente declarar os tipos das duas subárvores para que a prova tenha sucesso (tente omitir a declaração e veja se você consegue inferir o problema, examinando atentamente o estado da prova).

```
theorem th_refl01isA: "reflect (reflect t) = t"
  proof (induct t)
     show  "reflect (reflect Leaf) = Leaf" by simp
  next
     fix x0 and lt0::"'a Tree" and rt0::"'a Tree"
     assume IH1: "reflect (reflect lt0) = lt0"
     assume IH2: "reflect (reflect rt0) = rt0"
     show "reflect (reflect (Br x0 lt0 rt0)) = Br x0 lt0 rt0"
        by (simp add:IH1 IH2)
  qed

theorem th_reflect01isB : "reflect (reflect t) = t"
  proof (induct t)
   show "reflect (reflect Leaf) = Leaf"
     proof -
       have "reflect (reflect Leaf)=reflect Leaf" by (simp only: ref01)
       also  have "reflect Leaf = Leaf" by (simp only:ref01)
       finally show "?thesis" by simp
     qed
  next
     fix x0 and  lt0::"'a Tree" and  rt0::"'a Tree"
     assume IH1: "reflect (reflect lt0)=lt0"
     assume  IH2:"reflect (reflect rt0)=rt0"
     show "reflect (reflect (Br x0 lt0 rt0)) = (Br x0 lt0 rt0)"
     proof -
       have "reflect (reflect (Br x0 lt0 rt0)) =
        reflect (Br x0 (reflect rt0) (reflect lt0))" by (simp only:ref02)
       also have
       "...= Br x0 (reflect (reflect lt0)) (reflect (reflect rt0))"
           by (simp  only:ref02)
       also have "... = Br x0 lt0 rt0" by (simp only:IH1 IH2)
       finally show  "reflect (reflect (Br x0 lt0 rt0)) = Br x0 lt0 rt0 "
          by simp
     qed
  qed
```

5 Comentários Finais

Chegamos ao final deste pequeno e breve tutorial para `newbies`. Agora você precisa aplicar as técnicas vistas aqui e provar seus próprios teoremas. Introduzimos e listamos muitos deles durante as apresentações em sala de aula. A metodologia que usamos, aconselha a utilização do estilo procedural para exploração e descoberta de possíveis lemas auxiliares que serão necessários. Em um segundo momento, então, procuramos escrever a prova na linguagem Isar,

utilizando o nível de detalhe desejado. Para utilização do Isabelle como um verificador de provas (feitas, por exemplo, manualmente), sugiro a aplicação de passos atômicos de simplifcação, e é lógico, a utilização de tantos lemas quanto forem necessários para aumentar a legibilidade e a estrutura da prova como um documento formal. Em uma próxima versão, estaremos incorporando uma revisão de dedução natural (proof box style) e sua realização na linguagem Isabelle/Isar.

Let us not dream that reason can ever be popular.
Passions, emotions, can be made popular.
But reason remains ever a property of the few.

–Johann Wolfgang von Goethe (1749-1832)

end

Referências

[1] Robin Milner Michael Gordon and Christopher Wadsworth. Edinburgh LCF - A Mechanized Logic for Computation. In *Lectute Notes in Computer Science, vol. 78*. Springer-Verlag, 1980.

[2] Tobias Nipkow. A Tutorial Introduction to Structured Isar Proofs. Documentation available at the Isabelle website.

[3] L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1997.

[4] P. Rudnicki. An Overview of the Mizar Project. In *Workshop on Types for Proofs and Programs*, 1992.

[5] Jeremi Siek. Practical Theorem Proving with Isabelle/Isar. Lecture Notes, 2007.

[6] Lawrence Paulson Tobias Nipkow and Markus Wenzel. Isabelle/HOL – A Proof Assistant for Higher-Order Logic. In *Lectute Notes in Computer Science, vol. 2283*. Springer-Verlag, 2002.

[7] Markus Wenzel. The Isabelle/Isar Reference Manual. Documentation available at the Isabelle website.

[8] Markus Wenzel. *Isabelle/Isar – a versatile environment for human-readable formal proof documents*. PhD thesis, Institüt für Informatik, Technische Universität München, 2002.