Programming Languages meets Program Verification: The Chalmers University's Approach

Andrés Sicard-Ramírez

Logic and Computation Seminar
Universidad EAFIT
27 Feb 2007

Abstract

We shall give an overview of the CoVer project (Combining Verification Methods in Software Development) at Chalmers University, Sweden. The goal of this project is to provide an environment for Haskell programming which provides access to tools for automatic and interactive correctness proofs as well as to tools for testing. Moreover, we will show a short demo of two tools developed around CoVer project: Agda, a proof assistant using dependent type theory, and QuickCheck, a property based random testing tool for Haskell.

Sweden

- Area: 449.964 km^2. • Pop: 9.1 million • Capital: Stockholm

Gothenburg

- Area: 450 km^2. • Pop: 487.627

Independent Types

Definition (Independent types (abstract syntax))

$$V ::= v \mid V' \qquad \text{(type variables)}$$

$$C ::= c_1 \mid \cdots \mid c_n \qquad \text{(type constants)}$$

$$\begin{aligned}
\mathbb{T} ::=\ & V \\
\mid\ & C \\
\mid\ & \mathbb{T} \to \mathbb{T} && \text{(function types)} \\
\mid\ & \mathbb{T} \times \mathbb{T} && \text{(product types)} \\
\mid\ & \mathbb{T} + \mathbb{T} && \text{(disjoint union types)}
\end{aligned}$$

Independent Types

Example (Haskell's types)

- Type variables: `a,b,…`
- Type constants: `Int,Integer,Char,etc.`
- Function types: e.g. `Int` $\rightarrow$ `Int`
- Product types: e.g. `(Int,Char)`
- Disjoint union types: e.g.
  `data Sum a b = Inl a | Inr b`

(Untyped) Lambda-Calculus

Intuitively

λ-calculus element	Denotes
$\lambda x.x^2 + 1$ (abstraction)	Fn. $x \mapsto x^2 + 1$
$(\lambda x.x^2 + 1)3$ (application)	Fn. $x \mapsto x^2 + 1$ applied to 3
$(\lambda x.x^2 + 1)3 =_\beta 3^2 + 1$ (β-reduction)	The value of fn. $x \mapsto x^2 + 1$ applied to 3

Definition (λ-terms)

$$V ::= v \mid V' \quad \text{(variables)}$$

$$\Lambda ::= V \mid (\Lambda\Lambda) \mid (\lambda V \Lambda) \quad \text{(λ-terms)}$$

(Untyped) Lambda-Calculus

Definition (β-conversion)

$$(\lambda x.M)N =_\beta M[x := N] \quad \beta\text{-conversion}$$

Conventions

1. $x, y, z, \ldots$ denote variables
2. $M, N, L, \ldots$ denote λ-terms
3. $FM_1M_2 \ldots M_n$ denotes $(\ldots((FM_1)M_2)\ldots M_n)$ (application uses association to the left)
4. $\lambda x_1 \ldots x_n.M$ denotes $(\lambda x1(\ldots(\lambda x_n(M))\ldots))$ (abstraction uses association to the right)
5. Outermost parentheses are not written

(Untyped) Lambda-Calculus

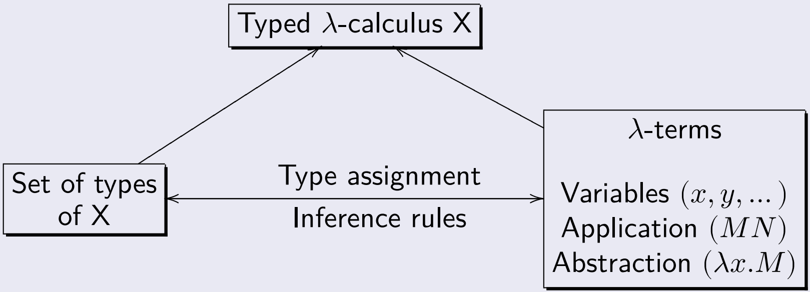
Examples

$$\begin{aligned} \mathrm{I} &\equiv \lambda x.x && \text{(identity function)} \\ \mathrm{K} &\equiv \lambda xy.x && \text{(first coordinate projection)} \\ \mathrm{S} &\equiv \lambda xyz.xz(yz) \end{aligned}$$

$$\begin{aligned} \mathrm{I}M &=_\beta M \\ \mathrm{K}MN &=_\beta M \end{aligned}$$

Typed Lambda-Calculus

General picture



Typed Lambda-Calculus

Definition (Simple typed λ-calculus (à la Curry))

Types $\mathbb{T}$:

type variables: $\alpha, \alpha', \dots \in \mathbb{T}$

function space types: $\sigma, \tau \in \mathbb{T} \Rightarrow (\sigma \to \tau) \in \mathbb{T}$

Inference rules:

$$\frac{}{x : \sigma \vdash x : \sigma} \text{(Axiom,Variable)}$$

$$\frac{\Gamma \vdash M : (\sigma \to \tau) \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau} (\to\text{-elimination, Application})$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x.M) : (\sigma \to \tau)} (\to\text{-introduction, Abstraction})$$

Typed Lambda-Calculus

Example (Proof in simple typed λ-calculus)

$$\cfrac{\cfrac{\cfrac{}{x : \sigma, y : \tau \vdash x : \sigma}\ (\text{Var})}{x : \sigma \vdash \lambda y.x : \tau \to \sigma}\ (\text{Abs})}{\vdash \lambda xy.x : \sigma \to \tau \to \sigma}\ (\text{Abs})$$

Example (Haskell)

```
k m n = m
k = \m n -> m

*GHCi> :t k
k :: a -> b -> a
```

Dependent Types

Definition (Dependent types)

"*A dependent type is a type that may depend on a value, typically like an array type, which depends on its length.*" [Barthe and Coquand 2002, p. 2]

Dependent Types

Definition (Set theory: Dependent function space)

Let $(B_x)_{x \in A}$ be an indexed family of sets. Then

$$\prod_{x \in A} B_x := \left\{ f : A \to \bigcup\nolimits_{x \in A} B_x \mid (\forall x \in A)(f(x) \in B_x) \right\}.$$

Note

If $B_x = B$ for all $x \in A$, then $\prod_{x \in A} B_x = A \to B$.

Definition (Type theory: Pi types)

$\prod_{x:A} B(x)$ is the type of terms f such that, for every $a : A$ then $f\ a : B(a)$.

Dependent Types

Definition (Set theory: Sum (disjoint union) of a family of sets)

Let $(B_x)_{x \in A}$ be an indexed family of sets. Then

$$\sum_{x \in A} B_x := \{\, (x, b) \mid x \in A \text{ and } b \in B_x \,\}.$$

Note

If $B_x = B$ for all $x \in A$, then $\sum_{x \in A} B_x = A \times B$.

Definition (Type theory: Sigma types)

$\sum_{x:A} B(x)$ is the type of pairs (M, N) such that $M : A$ and $N : B(M)$.

Constructive Interpretation of the Logical Constants

Definition

"*a proposition is defined by laying down what counts as proof of the proposition …a proposition is true if it has a proof, that is, if a proof of it can be given.*" [Martin-Löf 1984, p. 11]

Constructive Interpretation of the Logical Constants

a proof of the proposition	consist of (BHK-interpretation)	has the form
$A \wedge B$	a proof of A and a proof of B	(a, b), where a is a proof of A and b is a proof of B
$A \vee B$	a proof of A or a proof of B	$\mathrm{inl}(a)$, where a is a proof of A, or $\mathrm{inr}(b)$, where b is a proof of B
$\perp$	has not proof	
$A \supset B$	a method which takes any proof of A into a proof of B	$\lambda x.b(x)$, where $b(a)$ is a proof of B provided a is a proof of A
$(\forall x)B(x)$	a method which takes an arbitrary individual a into a proof of $B(a)$	$\lambda x.b(x)$, where $b(a)$ is a proof of $B(a)$ provided a is a proof of A
$(\exists x)B(x)$	an individual a and a proof of $B(a)$	(a, b), where a is an individual and b is a proof of $B(a)$

Curry-Howard Isomorphism

"*If we take seriously the idea that a proposition is defined by laying down how its canonical proofs are formed and accept that a set is defined by prescribing how its canonical elements are formed, then it is clear that it would only lead to unnecessary duplication to keep the notions of proposition and set…apart.*" [Martin-Löf 1984, p. 13]

Curry-Howard Isomorphism

A	$a : A$	
A is a set	a is an element of the set A	$A \neq \emptyset$
A is a proposition	a is a proof (construction) of the proposition A	A is true
A is a problem	a is a method of solving the problem A	A is solvable
A is a specification	a is a program than meets the specification A	A is satisfiable

Curry-Howard isomorphism (propositions-as-sets, formulas-as-types)

$A \wedge B = A \times B$	(product type)
$A \vee B = A + B$	(sum type)
$A \supset B = A \to B$	(function type)
$\bot = \mathrm{N}_0$	(empty type)
$\top = \mathrm{N}_1$	(unit type)
$\neg A = A \to \bot$	
$(\forall x)B(x) = \prod_{x:A} B(x)$	(Pi type)
$(\exists x)B(x) = \sum_{x:A} B(x)$	(Sigma type)

Curry-Howard Isomorphism

Example (Curry-Howard isomorphism working)

$\lambda \rightarrow$: simple typed λ-calculus
$IPC(\rightarrow)$: Implicational fragment of intuitionistic propositional logic

$$\dfrac{\dfrac{\dfrac{}{x : \sigma, y : \tau \vdash_{\lambda\rightarrow} x : \sigma}\,(\text{Var})}{x : \sigma \vdash_{\lambda\rightarrow} \lambda y.x : \tau \rightarrow \sigma}\,(\text{Abs})}{\vdash_{\lambda\rightarrow} \lambda xy.x : \sigma \rightarrow \tau \rightarrow \sigma}\,(\text{Abs}) \qquad \dfrac{\dfrac{\dfrac{}{\sigma, \tau \vdash_{IPC(\rightarrow)} \sigma}\,(\text{Ax})}{\sigma \vdash_{IPC(\rightarrow)} \tau \rightarrow \sigma}\,(\rightarrow\text{-intro})}{\vdash_{IPC(\rightarrow)} \sigma \rightarrow \tau \rightarrow \sigma}\,(\rightarrow\text{-intro})$$

Other Slides

Remark: The other slides shown in talk, that is to say, Prof. Dybjer's slides, can be found in http://www.cs.chalmers.se/~peterd/ under the "Combining testing and proving" link.

Acknowledgements

- We thank Peter Dybjer and Ana Bove (Chalmers University of Technology), Alberto Pardo (Universidad de la República), and Francisco Correa (Universidad EAFIT) for help us in all issues related to our visit to Chalmers University.
- We thank Universidad EAFIT and LerNET project for financial support.

References

Barthe, G. and Coquand, T. (2002). An Introduction to Dependent Type Theory. In: Applied Semantics. Ed. by Barthe, G., Dybjer, P., Pinto, L., and Saraiva, J. Vol. 2395. Lecture Notes in Computer Science, pp. 1–41. DOI: 10.1007/3-540-45699-6_1.

Martin-Löf, P. (1984). Intuitionistic Type Theory. Bibliopolis.