Martin-Löf Type Theory and the Logic of Functional Programs

Peter Dybjer

Amagasaki
11 May 2004

The Cover project – Combining Verification Methods in Software Development

- integrate a variety of verification methods
- smooth progression from hacking code to formal proofs of correctness
- hope to handle systems on a much larger scale than hitherto

This project builds on research at Chalmers in interactive theorem provers, formal methods, program analysis and transformation, and automatic testing. The functional language Haskell is used both as an implementation tool and a test-bed.

Existing tools

QuickCheck – a random testing tool for Haskell programs (Claessen, Hughes)

Automatic theorem provers for first order predicate logic

- translating Haskell programs to first order axioms (Claessen, Hamon)

Agda/Alfa – a proof assistant for Martin-Löf type theory with

- QuickCheck-like testing tool (Qiao)
- Agsy - automatic proof search tool (Lindblad)

The Cover dilemma

External logic (automatic subgroup) First order predicate logic with axioms for functional programs – we have automatic theorem provers!

- Logic of general recursive, lazy functional programs (LTC = Logical Theory of Constructions $\approx$ first order theory of combinators)

Integrated logic (interactive subgroup) Martin-Löf type theory – we have a proof assistant!

- Logic of primitive recursive programs (higher type, structural recursion on arbitrary inductive type)
- Generalizing primitive recursion? Encoding general recursion, encoding lazy data structures?

Some questions

- How can we use an automatic theorem prover for predicate logic to prove theorems in Martin-Löf type theory?
- What axioms for functional programs do we need?
- How can we view Martin-Löf type theory as a subsystem of LTC?
- How can we view LTC as a subsystem of Martin-Löf type theory?
- Foundations: Martin-Löf type theory vs LTC.

How can we use an automatic theorem prover for predicate logic to prove theorems in Martin-Löf type theory?

Interpretation of Heyting arithmetic in Martin-Löf type theory

$$
\begin{aligned}
(P \supset Q)^* &= P^* \to Q^* \\
(P \wedge Q)^* &= P^* \times Q^* \\
(P \vee Q)^* &= P^* + Q^* \\
\top^* &= \mathbf{1} \\
\bot^* &= \emptyset \\
(\forall x.P)^* &= \Pi x : \mathrm{N}.P^* \\
(\exists x.P)^* &= \Sigma x : \mathrm{N}.P^* \\
(a = b)^* &= \mathrm{E}\,(a^*, b^*)
\end{aligned}
$$

Interpret $0, \mathrm{Succ}, +, *$ as the obvious operations on N.

Quantification over other types ...

An Agda goal:

$$? \in \Pi xs : \mathrm{ListN}.\, \mathrm{Sorted}(\mathrm{sort}\, xs)$$

We would like to translate this into *typed* predicate logic formula

$$\forall xs : \mathrm{ListN}.\, \mathrm{Sorted}(\mathrm{sort}\, xs)$$

which can be simulated by the untyped predicate logic formula

$$\forall xs.\, \mathrm{ListN}(xs) \supset \mathrm{Sorted}(\mathrm{sort}\, xs)$$

so that we can call the first order theorem prover.

Three ways to interpret $a : A$ in predicate logic

- as a formula A^* using Curry-Howard, provided we are only interested in the truth of A, and not the proof object a. Restrictions?

- as $A^*[a^*]$, that is, types are interpreted as unary predicates, qua formulas with one free variable. Terms are interpreted as first order terms. (Aczel 1974, etc, see next page)

- as $a^* \in A^*$, that is, $\in$ is a binary predicate symbol, and terms and types are interpreted as first order terms. (Tammet and Smith 1996).

What axioms for functional programs do we need?
First order theory of combinators.
Towards a first order theory of Haskell programs.
–
How can we view Martin-Löf type theory as a subsystem?

Interpretations of Martin-Löf type theory in predicate logic

- Aczel 1974 (first order theory of combinators), 1980 (Frege structures), 1990 (logical theory of constructions)
- Smith 1978, 1984 (type-free theory of propositions)
- Martin-Löf, various unpublished notes 1983-85 (basic logical theory)

Also realizability interpretation of Beeson 1982.

Combining logic and lambda calculus – variations

Scott, Milner et al LCF, PPλ.

Feferman (explicit mathematics), Hayashi (PX), Sato, Tatsuta, Fourman, Beeson (lambda logic)...

Aczel: different approaches to LTC (first order vs higher order, etc.)

- recursion-theoretic (Feferman, ...)
- denotational (Aczel 1974, 1980, Smith 1978, 1984)
- operational (Martin-Löf 1983, Aczel 1983, 1991)

First order theory of combinators (Aczel 1974)

Terms (one binary function symbol + two constants)

$$t ::= x \mid t\,t \mid \mathrm{K} \mid \mathrm{S}$$

Propositions (three unary predicate symbols + equality + logical constants)

$$\Phi ::= \mathcal{N}(t) \mid \mathcal{P}(t) \mid \mathcal{T}(t) \mid t = t \mid \forall x.\Phi \mid \exists x.\Phi \mid \Phi \supset \Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \top \mid \bot$$

First order theory of combinators ...

- $s = t$ means that s and t are convertible:

$$\begin{aligned} \mathrm{K}\, x\, y &= x \\ \mathrm{S}\, x\, y\, z &= x\, z\, (y\, z) \end{aligned}$$

- $\mathcal{N}(t)$ means that t is equal to a Church numeral (λ-terms by bracket abstraction). The rules are

$$\begin{aligned} &\mathcal{N}(0) \\ \mathcal{N}(x) &\supset \mathcal{N}(\mathrm{Succ}\, x) \\ \Phi[0] &\supset (\forall x.\, \Phi[x] \supset \Phi[\mathrm{Succ}\, x]) \supset \forall y.\, \mathcal{N}(y) \supset \Phi[y] \end{aligned}$$

Internal propositions and truths

- $\mathcal{P}(t)$ means that t is a code for a proposition. Such codes (internal propositions) are also obtained by Church-style encodings.

- $\mathcal{T}(t)$ means that t is a code for a true proposition.

The interpretation of Martin-Löf type theory in Aczel's first order theory of combinators

Two examples:

$$f : \mathrm{N} \to \mathrm{N} \quad \text{as} \quad \forall x.\, \mathcal{N}(x) \supset \mathcal{N}(f\, x)$$
$$c : \mathrm{N} \times \mathrm{N} \quad \text{as} \quad \exists x.\, \exists y.\, \mathcal{N}(x) \wedge \mathcal{N}(y) \wedge c = (x, y)$$

Church pairs: $(t, t') = \lambda x.\, x\, t\, t'$.

Interpretation of hypothetical judgements in Martin-Löf type theory

$$x_1 : A_1, \ldots, x_n : A_n[x_1, \ldots, x_{n-1}] \vdash a[x_1, \ldots, x_n] : A[x_1, \ldots, x_n]$$

is interpreted as

$$A_1^*[x_1], \ldots, A_n^*[x_1, \ldots, x_{n-1}, x_n] \vdash A^*[x_1, \ldots, x_n, a^*[x_1, \ldots, x_n]]$$

Interpretation of types

$$
\begin{aligned}
\mathrm{N}^*[z] &:= \mathcal{N}(z) \\
(\Pi x : A.B[x]])^*[z] &:= \forall x.A^*[x] \supset B^*[x, z\,x] \\
(\Sigma x : A.B[x]])^*[z] &:= \exists x.\exists y.A^*[x] \wedge B^*[x,y] \wedge z = (x,y) \\
(A+B)^*[z] &:= (\exists x.A^*[x] \wedge z = \mathrm{Inl}\,x) \vee (\exists y.B^*[y] \wedge z = \mathrm{Inr}\,y) \\
\mathrm{U}^*[z] &:= \forall x.\mathcal{P}(z\,x) \\
\mathrm{T}(a)^*[z] &:= \mathcal{T}(a^*\,z)
\end{aligned}
$$

Church-style encodings of pairs and injections. Universes are given à la Tarski.

Integrated vs external logic of functional programs: pros and cons

Integrated (MLTT): based on Curry-Howard (proofs as programs) Elegant normalizing type checkers; notations for proofs (modules, etc)

External (LTC): based on predicate logic (a "logical comment") and lambda calculus. More flexible, direct (even "canonical") approach for reasoning about general recursive programs (Dybjer 1985), can be extended with greatest fixed point rules (Dybjer and Sander 1989), and domain logic rules (LCF).

Reasoning about general recursive programs

Introduce a fixed point combinator

```
fix f = f (fix f)
```

We can now program the Euclidean division algorithm

```
div i j = if iT < j then 0 else Succ (div (i - j) j)
```

```
div = fix (\f i j -> if i < j then 0 else Succ (f (i - j) j))
```

and prove it correct.

Specification of division

$$\forall i.\, \forall j.\, \mathcal{N}(i) \supset \mathcal{N}(j) \supset j > 0 \supset \mathrm{DIV}(i, j, \mathrm{div}\, i\, j)$$

where

$$\mathrm{DIV}(i, j, q) = \exists r.\, \mathcal{N}(r) \supset r < j \supset q * j + r = i$$

QuickCheckable specification

$$\mathrm{DIV}(i, j, q) = q * j \leq i < (\mathrm{Succ}\, q) * j$$

How can we view LTC as a subsystem of Martin-Löf type theory?

Interpretation of LTC in Martin-Löf type theory

We here take LTC = first order theory of combinators of Aczel 1974.

Introduce inductively defined set

$$\mathrm{Term} : \mathbf{Set}$$

of combinatory terms with constructors:

$$\begin{array}{rcl} \hat{\mathrm{K}} & : & \mathrm{Term} \\ \hat{\mathrm{S}} & : & \mathrm{Term} \\ (@) & : & \mathrm{Term} \to \mathrm{Term} \to \mathrm{Term} \end{array}$$

Introduce inductively defined relation

$$(\mathrel{\hat{=}}) : \mathrm{Term} \to \mathrm{Term} \to \mathbf{Set}$$

Constructors:

$$\begin{array}{rcl}
\mathrm{ref} & : & (a : \mathrm{Term}) \to a \mathrel{\hat{=}} a \\
\mathrm{sym} & : & (a, b : \mathrm{Term}) \to a \mathrel{\hat{=}} b \to b \mathrel{\hat{=}} a \\
\mathrm{tra} & : & (a, b, c : \mathrm{Term}) \to a \mathrel{\hat{=}} b \to b \mathrel{\hat{=}} c \to a \mathrel{\hat{=}} c \\
\mathrm{cong} & : & (c, a, c', a' : \mathrm{Term}) \to c \mathrel{\hat{=}} c' \to a \mathrel{\hat{=}} a' \to c@a \mathrel{\hat{=}} c'@a' \\
\mathrm{Kax} & : & (a, b : \mathrm{Term}) \to \hat{\mathrm{K}}\,@a@b \mathrel{\hat{=}} a \\
\mathrm{Sax} & : & (a, b, c : \mathrm{Term}) \to \hat{\mathrm{S}}\,@a@b@c \mathrel{\hat{=}} a@c@(b@c)
\end{array}$$

Introduce inductively defined predicate

$$\hat{\mathcal{N}} : \mathrm{Term} \to \mathbf{Set}$$

Constructors:

$$\begin{aligned} 0\mathrm{ax} &: \hat{\mathcal{N}}\ \hat{0} \\ \mathrm{Succax} &: (a : \mathrm{Term}) \to \hat{\mathcal{N}}\ a \to \hat{\mathcal{N}}\ (\hat{\mathrm{Succ}}@a) \\ \mathrm{subst} &: (a, b : \mathrm{Term}) \to a \mathrel{\hat{=}} b \to \hat{\mathcal{N}}\ b \to \hat{\mathcal{N}}\ a \end{aligned}$$

Adding inductive and coinductive predicates to LTC

So far only one inductively defined predicate: $\mathcal{N}(t)$ – to be a *total*, *finite* natural number.

We would like to extend LTC with

- inductively defined predicates in general (not only $\mathcal{N}$)
- coinductively defined predicates in general

It is possible to formulate an intuitionistic theory for this, but if we use classical logic we can employ the μ-calculus of Hitchcock and Park 1973 (also de Bakker and Scott ca 1970) together with a first order theory of combinators (Dybjer and Sander 1989).

The Programatica project

The P-logic of the Programatica project (Kieburtz and Harrison 2003) looks related.

What is the μ-calculus of Hitchcock and Park?

Add predicates

$$\mu X.P$$

to the syntax of first order predicate logic, provided X only occurs positively in P! (Formally, we leave first order logic and introduce a new syntactic category of n-place predicate terms P, add n-place predicate variables X and abstracts $\{x_1, \ldots, x_n \mid \Phi\}$, etc).

The meaning of $\mu X.P$ is the least fixed point of the monotone function which maps the meaning of X to the meaning of P.

An example in the μ-calculus of combinators

$$\mathcal{N} \equiv \mu X.\{x \mid x = 0 \vee \exists y.X(y) \wedge x = \mathrm{Succ}\, y\} \equiv \mu X.\{0\} \cup \mathrm{Succ}\, X$$

Greatest fixed points in the μ-calculus

Greatest fixed points can be defined in terms of least fixed points

$$\nu X.P \equiv \mathcal{C}(\mu X.\mathcal{C}(P[\mathcal{C}X/X]))$$

For example, natural numbers with infinity;

$$\mathcal{N}^{\infty} = \nu X.\{x \mid x = 0 \vee \exists y.X(y) \wedge x = \mathrm{Succ}\, y\} \equiv \nu X.\{0\} \cup \mathrm{Succ}\, X$$

Infinite lists and bisimilarity

Infinite lists of finite total natural numbers

$$\begin{aligned} \mathrm{ListN}^{\infty} &= \nu X.\{xs \mid \exists x, ys.\, X(ys) \wedge xs = \mathrm{Cons}\, x\, ys\} \\ &= \nu X.\mathrm{Cons}\,\mathrm{N}\,X \end{aligned}$$

Bisimilarity of infinite lists

$$\approx \equiv \nu X.\mathrm{Cons}(=_{\mathrm{N}})X$$

where $=_{\mathrm{N}}$ is equality of finite total natural numbers, and Cons is an operation on binary relations here.

Reasoning about infinite lists – an example

Program. Alternating bit protocol written as a stream processing Miranda program using the idea of Kahn dataflow network. Non-determinism of transmission channels modelled by oracle streams.

Specification. If the input is an infinite stream of data values, and the oracle streams are fair (in a certain sense), then the output stream is bisimilar to the input stream.

Proof. Implementation of the μ-calculus in Isabelle. Proof uses induction, coinduction and conversion.

For details, see P. Dybjer and H. Sander, Formal Aspects of Computing (1989) 1: 303-319: A functional programming approach to the specification and verification of concurrent systems.

First order theorem proving for the μ-calculus

The μ-calculus is not a first order theory. However, we can nevertheless use a first order logic prover as follows. Replace each occurrence of $\mu X.\Phi$ with a new predicate symbol, and the axiom schemas for least fixed points by their instances.

Conclusion

What are the relevant first order axioms for Haskell? The μ-calculus of combinators? Yes, this gives a good logic for a pure functional programming language with recursive datatypes, including rules for induction and coinduction. But ...

- We may want to add new axioms, eg from domain theory. In general add principles which are true in the standard domain theoretic model.

- Many issues concerning "syntactic sugar" (pattern matching, Haskell's class system, ...). Some of these are addressed by P-logic. What is the semantics of Haskell?

LTC vs Martin-Löf type theory as foundation

The Brouwer-Heyting-Kolmogorov interpretation of the intuitionistic connectives

A proof of $A \supset B$ is a method which transforms a proof of type A to a proof of type B.

A proof of $A \wedge B$ is a pair consisting of a proof of A and a proof of B.

A proof of $A \vee B$ is either a proof of A or a proof of B.

Martin-Löf's meaning explanations for the connectives

$c : A \supset B$ iff $c \Rightarrow_\beta \lambda x.\, b$ and $b[x := a] : B$ for all $a : A$.

$c : A \wedge B$ iff $c \Rightarrow_\beta (a, b)$ and $a : A$ and $b : B$.

$c : A \vee B$ iff either $c \Rightarrow_\beta \mathrm{Inl}\, a$ and $a : A$, or $c \Rightarrow_\beta \mathrm{Inr}\, b$ and $b : B$.

Brouwer-Heyting-Kolmogorov interpretation of the intuitionistic quantifiers

A proof of $\forall x : A. B$ is a method which for an arbitrary element a of A returns a proof of $B[x := a]$.

A proof of $\exists x : A. B$ is a pair consisting of a element a of A (the witness) and a proof of $B[x := a]$.

Martin-Löf's meaning explanations for the quantifiers

$c : \forall x : A.\, B$ iff $c \Rightarrow_\beta \lambda x.\, b$ and $b[x := a] : B[x := a]$ for all $a : A$.

$c : \exists x : A.\, B$ iff $c \Rightarrow_\beta (a, b)$ and $a : A$ and $b : B[x := a]$.

The correctness of all rules of intuitionistic typed predicate logic can be justified by Martin-Löf's meaning explanations.