Stanford LCF[*]

Gergely Buday[a]

[a]Eszterházy Károly University
Gyöngyös
buday.gergely@uni-eszterhazy.hu

Abstract

In this paper I describe Stanford LCF and its historical significance. It is based on Dana Scott's Logic of Computable Functions. It is one of the earliest theorem prover exhibiting ideas that are prevalent since in the family of LCF-style provers. Its descendants Edinburgh LCF, Cambridge LCF, the HOL family of theorem provers and Isabelle/HOL are all important developments in theorem proving history. It was created almost fifty years ago so it is time to give an account on it.

Keywords: functional programming, theorem proving, contemporary history

MSC: 68N18, 68T15, 01A65

Introduction

Fifty years ago, Dana Scott wrote a paper on a model of the lambda-calculus that remained unpublished until 1993 [8] — the cause for not publishing it was a remark in the paper on the type-free lambda calculus not having a model but soon Scott found a model for it. Based on this work, Robin Milner wrote a machine implementation of it [6].

The logic

Scott gave his LCF as logic of typed combinators which Milner translated into the typed λ-calculus [5]. He wrotes that λ-calculus is an easier formalism to use but its metatheory is complicated because of bound variables — this is still an active research area in programming language formalisation [7] [10] [1] [4].

[*]This work was supported by the ”Complex Development of Research Capacities at the Eszterházy Károly University” EFOP-3.6.1-16-2016-00001 project.

The implementation

Milner wrote a user's manual for Stanford LCF, a proof-checking program as he defines it [6]. He expressed gratitude to John McCarthy *for encouraging me to undertake this work.* This happened in 1972, and as early as that he envisioned his system to generate formal proofs about not only integers and lists but about computer programs and their semantics. Already appear in the system two components that are present in today's LCF-style theorem provers: the subgoaling facility and the simplification mechanism.

The logic LCF is presented using the typed λ-calculus, not the typed combinators S and K as Scott originally expressed it. Milner uses an axiomatization for integers by Scott and introduces a partial axiomatization for lists. Besides these, he writes that LCF is expressive enough to prove e.g. the equivalence of recursion equation schemata without introducing non-logical axioms.

The language implemented

The basic types are *tr* and *ind*, the type of truth values and of individuals, the latter is still used in HOL4 [3] and it originates from [2]. Terms can be

- identifiers — alphanumeric strings
- applications $s(t)$, where $s : \beta_1 \to \beta_2$ and $t : \beta_1$
- conditionals $(s \to t_1, t_2) : \beta$, where $s : tr$ and $t_1, t_2 : \beta$
- λ-expressions $[\lambda x.s] : \beta_1 \to \beta_2$, where $x : \beta_1$ and $s : \beta_2$
- α-expressions $[\alpha x.s] : \beta$, where $x, s : \beta$

For identifiers, the document writes *"we assume that the type of each identifier is uniquely determined in some manner"*. More to that, later *"note that in the machine implementation there is no type-checking whatsoever, we rely on the user to use types consistently"* is added. $[\alpha x.s] : \beta$ is the least (minimal in the original text) fixed point of the function $[\lambda f.s]$ as follows:

$$[\alpha f.[\lambda x.(p(x) \to f(a(x)), b(x))]]$$

creates the least fixed point for the following function definition:

$$f(x) = if\ p(x)\ then\ f(a(x))\ else\ b(x)$$

TT and FF are true and false, while UU is the totally undefined object of any type.

An atomic well-formed formula is $s \subset t$, where s and t have the same type. Its meaning is that t is as well defined as s and is consistent with s. Lists of atomic well-formed formulae form well-formed formulae, and they are meant as the conjunction of their element atomic well-formed formulae. At the end, sentences are implications of well-formed formulae: $P \vdash Q$.

A proof is a series of sentences, where each is derived from previous sentences by an inference rule.

Inference rules

In the following rules, $P(s/x)$ is the substitution of s into all free occurrences of x in P, using α-conversion in P to avoid free variable caption.

Derivability rules: $\vdash$

$$\text{INCL}\ \frac{}{P \vdash Q}\ \text{(Q is a subset of P)}$$

$$\text{CONJ}\ \frac{P \vdash Q_1 \qquad P \vdash Q_2}{P \vdash Q_1 \cup Q_2}$$

$$\text{CUT}\ \frac{P_1 \vdash P_2 \qquad P_2 \vdash P_3}{P_1 \vdash P_3}$$

Rules for better defined terms: $\subset$

$$\text{APPL}\ \frac{}{s_1 \subset s_2 \vdash t(s_1) \subset t(s_2)}$$

$$\text{REFL}\ \frac{}{P \vdash s \subset s}$$

$$\text{TRANS}\ \frac{P \vdash s_1 \subset s_2 \qquad P \vdash s_2 \subset s_3}{P \vdash s_1 \subset s_3}$$

UU rules

$$\text{MIN1}\ \frac{}{\vdash UU \subset s}$$

$$\text{MIN2}\ \frac{}{\vdash UU(s) \subset UU}$$

Conditional rules

$$\text{CONDT}\ \frac{}{\vdash TT \to s, t \equiv s}$$

$$\text{CONDU}\ \frac{}{\vdash UU \to s, t \equiv UU}$$

$$\text{CONDF}\ \frac{}{\vdash FF \to s, t \equiv t}$$

λ rules

$$\text{ABSTR}\ \frac{P \vdash s \subset t}{P \vdash [\lambda x.s] \subset [\lambda x.t]}\ x \text{ is not free in } P$$

$$\text{CONV}\ \frac{}{\vdash [\lambda x.s](t) \equiv s(t/x)}$$

$$\text{ETACONV}\ \frac{}{\vdash [\lambda x.y(x)] \equiv y}\ x \text{ and } y \text{ are distinct}$$

Truth rule

$$\text{CASES}\ \frac{P, s \equiv TT \vdash Q \qquad P, s \equiv UU \vdash Q \qquad P, s \equiv FF \vdash Q}{P \vdash Q}$$

Fixed point α rules

$$\text{FIXP}\ \frac{}{\vdash [\alpha x.s] \equiv s([\alpha x.s]/x)}$$

$$\text{INDUCT}\ \frac{P \vdash Q(UU/x) \qquad P, Q \vdash Q(t/x)}{P \vdash Q([\alpha x.t]/x)}\ x \text{ is not free in } P$$

Commands

There are four groups of commands in Stanford LCF:

- rules of inference
- goal oriented commands to state and process goals and subgoals
- system commands for displaying and handling proofs
- defining axioms and theorems and to recall and instantiate them

The SIMPSET command allows to add or remove simplification rules from the simplification set, still present in current LCF-style theorem provers. SHOW does what its name suggests: shows subgoals, *steps* in Stanford LCF parlance. FETCH gets axioms and theorems from files. The system description says *"much of the user types is dependent on the stepnumbers that the system is generating, so the use of files prepared offline is limited"*. There is the INFIX command making an identifier used as infix, showing that already in Stanford LCF the developers thought of making the syntax close to the mathematical practice, aiming user-friendliness. With LABEL one could avoid the use of stepnumbers.

With the AXIOM command one can define axioms. The THEOREM command allows the user to define theorems and optionally to name the axioms it depends on. Mathematically this is an interesting feature as the searching of what axioms are needed to prove a theorem is now a research field of its own [9]. The system checked whether the axioms mentioned are present in the proof. The USE command also checks the axioms of a theorem whether they are available and treats the theorem as a meta-theorem where its free variables are to be instantiated, another useful concept already introduced in Stanford LCF and present in modern LCF-style theorem provers.

Tactics

Tactics, which are available in today's HOL4 theorem prover already appear in Stanford LCF. E.g. CONJ breaks down a well-formed formula into its constituents — remember, a wff is a conjunction of its element atomic wffs. CASES does the backward step of the CASES rule: for a term s and wff P the system generates the three subgoals $P\ SASSUME\ s \equiv TT$, $P\ SASSUME\ s \equiv UU$ and $P\ SASSUME\ s \equiv FF$.

SIMPL does simplification generating a new subgoal. SUBST substitutes the right hand side of a subgoal for the left hand side of another one. INDUCT applies the similarly named rule backwards: if the subgoal is a recursive definition $s \equiv [\alpha y.t]$ then it creates two subgoals $P(UU/s)$ and $P(t(y'/y)/s)\ ASSUME\ P(y'/s)$ where

Simplification

Some more detail about simplification as it is interesting historically. The system description reminds the user not to introduce simplification rules that results in an infinite rewriting. Allowing the use of assumptions as simplification rules is already present. Recursive definitions are handled smartly: although such a definition could lead to an indefinite expansion, with the use of other members of the simplification set termination can be achieved.

References

[1] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A type and scope safe universe of syntaxes with binding: Their semantics and proofs. *Proc. ACM Program. Lang.*, 2(ICFP):90:1–90:30, July 2018.

[2] Alonzo Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.

[3] HOL Developers. The HOL System DESCRIPTION. Technical report, June 2018.

[4] Lorenzo Gheri and Andrei Popescu. A formalized general theory of syntax with bindings: Extended version. *Journal of Automated Reasoning*, Apr 2019.

[5] Robin Milner. Implementation and applications of scott's logic for computable functions. In *Proceedings of ACM Conference on Proving Assertions About Programs*, pages 1–6, New York, NY, USA, 1972. ACM.

[6] Robin Milner. Logic for Computable Functions: Description of a Machine Implementation. Technical report, Stanford, CA, USA, 1972.

[7] A. M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*, volume 57 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2013.

[8] Dana S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121(1):411 – 440, 1993.

[9] J. Stillwell. *Reverse Mathematics: Proofs from the Inside Out.* Princeton University Press, 2019.

[10] Christian Urban and Cezary Kaliszyk. General Bindings and Alpha-Equivalence in Nominal Isabelle. *Logical Methods in Computer Science*, 8(2:14), 2012.