Type theory and functional programming

This talk will about the connections between type theory and functional programming

Can we see type theory as a functional programming language?

Some basic questions still need to be clarified

Type theory as a functional programming language

The interest in having a programming language *integrated* to a proof system is perfectly illustrated by the work of G. Gonthier (2004) on the complete formal proof of the four color theorem

Gonthier clarifies/simplifies the C programs used in the proof of Robertson et al. by rewriting them as functional programs (with proofs of correctness)

One can *internalize* decision procedures (represented as functional programs) and Gonthier uses systematically the technique of *reflection*

Asp

Asp was designed by Bengt Nordström in January 1980 to "show the similarity between type theory and other functional languages. **Asp** can be described as a programming language with general recursion and no dependent types. The treatment of types as objects is different from common functional languages."

It appeared in the Programming Methodology Group Report 1
Description of a Simple Programming Language, April 1984

Type Theory

Suppose we have defined a function which to an arbitrary object x of type A assigns a type $B(x)$. Then the cartesian product

$$(\Pi x \in A)B(x)$$

is a type, namely the type of functions which take an arbitrary object x of type A into an object of type $B(x)$.

Type Theory

Functions may be introduced by *explicit definition*. That is, if we build up a term from constants for already defined objects and a variable x that denotes an arbitrary object of type A and if this term t denotes a term of type $B(x)$, then we may introduce a function f of type $(\Pi x \in A)B(x)$ be means of the schema

$$f(x) = t$$

where explicit mention of parameters is suppressed.

Type Theory

We can introduce the type N, the type of *natural numbers*.

0 is an object of type N and, if x is an object of type N, so is its successor $S(x)$.

Type Theory

Given an object c of type $C(0)$ and a function g of type

$$(\Pi x \in N)C(x) \to C(S(x))$$

we may introduce a function f of type $(\Pi x \in N)C(x)$ by the *recursion* schema

$$f(0) = c \qquad\qquad f(S(x)) = g(x, f(x))$$

Type Theory

Thinking of $C(x)$ as a proposition f is a proof of the universal proposition $(\Pi x \in N)C(x)$ which we get by applying the principle of *mathematical induction*

In the case $C(x)$ does not depend explicitely on x we get the schema of primitive recursion (at higher types), schema introduced by Hilbert and used later by Gödel

Type Theory

We can introduce the type Ord, the type of *ordinal numbers*.

0 is an object of type Ord and, if x is an object of type Ord, so is its successor $S(x)$ and if u is a function of type $N \to Ord$ then its limit $L(u)$ is an object of type Ord

Type Theory

Given an object c of type $C(0)$ and a function g of type

$$(\Pi x \in Ord)C(x) \to C(S(x))$$

and h a function of type

$$(\Pi u \in N \to Ord)((\Pi x \in N)C(u(x)) \to C(L(u))$$

we may introduce a function f of type $(\Pi x \in Ord)C(x)$ by the *recursion* schema

$$f(0) = c \qquad f(S(x)) = g(x, f(x)) \qquad f(L(u)) = h(u, f \circ u)$$

where $(f \circ u)(x) = f(u(x))$

Type Theory

Thinking of $C(x)$ as a proposition, f is a proof of the universal proposition $(\Pi x \in Ord)C(x)$ which we get by applying the principle of *transfinite induction* over the second number class ordinals.

Type Theory

In the formal theory the abstract entities (natural numbers, ordinals, functions, types, and so on) become represented by certain symbol configurations, called terms, *and the definitional schema, read from the left to the right, become mechanical* reduction rules *for these symbol configurations.*

Type theory *effectuates the computerization of abstract intuitionistic mathematics that above all Bishop has asked for*

It provides a framework in which we can express *conceptual* mathematics in a *computational* way.

How to implement type theory? How to do actual computations?

Asp: types as objects

Asp has *labelled sum* types

If $T_1, \ldots, T_n$ are types then so is $T = c_1\ T_1 + \cdots + c_n\ T_n$

If e is an object of type T_i then $c_i\ e$ is an object of type T

An object in canonical form of type T is of the form $c_i\ e$ where e is an object of type T_i

Asp: types as objects

Asp has a type of (small) types U

We can explain the type N as a recursively defined object of type U

$$N : U = 0\ () + S\ N$$

then $S\ x$ is of type N if x is of type N

Terminating general recursion

Theorem 1 (1987) *All iterating constructs in type theory can be reduced to pattern matching and the general recursion operator*

Pattern matching: if $T = c_1\ T_1 + \cdots + c_n\ T_n$ we can define an object of type $(\Pi x \in T)C(x)$ by the equations

$$f\ (c_1\ x) = e_1 \qquad \ldots \qquad f\ (c_n\ x) = e_n$$

provided e_i is of type $C(c_i\ x)$

Terminating general recursion

For instance, the primitive recursive operator of Gödel system T can be defined recursively using pattern matching

$$R\ 0 = a \qquad\qquad R\ (S\ n) = g\ n\ (R\ n)$$

where a and g are parameters of the definition

Variation: LML

Lazy ML *Compiling Lazy Functional Languages, Part II* (1987) L. Augustsson

Telescopes: vector of types

$T = c_1\ \vec{T_1} + \cdots + c_n\ \vec{T_n}$

$f\ (c_1\ \vec{x_1}) = e_1 \qquad \ldots \qquad f\ (c_n\ \vec{x_n}) = e_n$

provided e_i is of type $C(c_i\ \vec{x_i})$ in the context $\vec{x_i} : \vec{T_i}$

$N = 0 + S\ N$ and 0 of type N and $S\ x$ of type N if x of type N

Representation of primitive recursion

Given an object c of type $C(0)$ and a function g of type

$$(\Pi x \in N)C(x) \rightarrow C(S(x))$$

we may introduce a function f of type $(\Pi x \in N)C(x)$ by the recursion schema

$$f\ 0 = c \qquad\qquad f\ (S\ x) = g\ x\ (f\ x)$$

f is a function defined recursively by pattern matching

Representation of type theory

In order to represent all iterating constructs of type theory it is enough to have a programming language with

a type of (small) types U

labelled sum

recursive definitions (of types and functions)

Type theory as a functional programming language

The language **Asp** points out how the representation of type theory as a functional programming language should look like

It has a simple description (one page), a simple interpreter (in itself), a simple operational semantics (evaluation rules) and a simple denotation semantics

It does not have dependent types, and does not explain how to compare types

A Simple Progamming Language

Programs

$$M, A \;::=\; x \mid M\ M \mid \lambda x.M \mid M\ D \mid c\ \vec{M} \mid B \mid L$$

Branches, Labelled Sums and (recursive) Definitions

$$B \;::=\; c_1\ \vec{x_1} \to N_1, \ldots, c_l\ \vec{x_l} \to N_l \qquad L \;::=\; c_1\ T_1, \ldots, c_l\ T_l$$

$$D \;::=\; [\vec{x} : T = \vec{M}] \qquad T \;::=\; () \mid (x : A, T)$$

Environments and Values

Environments and Values

$$\rho, \sigma \ ::= \ () \mid \rho, x = u \mid D\rho \qquad u \ ::= \ M\rho \mid u\ u \mid x$$

Operational semantics (cf. eval in LISP)

$$(c\ \vec{M})\rho \rightarrow c\ (\vec{M}\rho)$$

$$(M_1\ M_2)\rho \rightarrow M_1\rho\ (M_2\rho) \qquad (M\ D)\rho \rightarrow M(D\rho)$$
$$(\lambda x.N)\rho\ u \rightarrow N(\rho, x = u) \qquad B\rho\ (c_i\ \vec{u}) \rightarrow N_i(\rho, \vec{x_i} = \vec{u})$$

where $B = c_1\ \vec{x_1} \rightarrow N_1, \ldots, c_l\ \vec{x_l} \rightarrow N_l$

Evaluation rules

Access rules

$$x(\sigma, x = u) \to u \quad x(\sigma, y = u) \to x\sigma$$

$$x\rho \to x(\sigma, \vec{x} = \vec{M}\rho)$$

where $\rho = [\vec{x} : T = \vec{M}]\sigma$

Examples

$$N = 0 + S\ N \qquad add = \lambda x.(0 \to x,\ S\ y \to S\ (add\ x\ y))$$

$$add\ x\ 0 = x \qquad add\ x\ (S\ y) = S\ (add\ x\ y)$$

$$Ord = 0 + S\ Ord + L\ (N \to Ord)$$

$$add = \lambda x.(0 \to x,\ S\ y \to S\ (add\ x\ y),\ L\ u \to L\ (\lambda n.add\ x\ (u\ n)))$$

$$add\ x\ 0 = x \qquad add\ x\ (S\ y) = S\ (add\ x\ y) \qquad add\ x\ (L\ u) = L\ (\lambda n.add\ x\ (u\ n))$$

Examples

$$N_0 = () \qquad N_1 = 0 \qquad N_2 = 0 + 1$$

$$T : N_2 \to U = (0 \to N_0,\ 1 \to N_1)$$

which corresponds to the equations

$$T\ 0 = N_0 \qquad\qquad T\ 1 = N_1$$

Examples

U for the type of (small) types

$\Pi\ A\ (\lambda x.B)$ for $(\Pi x \in A)B$

$A \rightarrow B$ if x not free in B

$N : U = 0 + S\ N$

Examples

$$eq_N : N \to N \to N_2 = (0 \to (0 \to 1,\ S\,y \to 0),\ S\,x \to (0 \to 0,\ S\,y \to eq_N\;x\;y))$$

$$eq_N\;0\;0 = 1 \qquad eq_N\;0\;(S\;y) = 0 \qquad eq_N\;(S\;x)\;0 = 0 \qquad eq_N\;(S\;x)\;(S\;y) = eq_N\;x\;y$$

$$(<) : N \to N \to N_2 = (0 \to (0 \to 0,\ S\ y \to 1),\ S\ x \to (0 \to 0,\ S\ y \to x < y))$$

$$0 < 0 = 0 \qquad 0 < (S\ y) = 1 \qquad (S\ x) < 0 = 0 \qquad (S\ x) < (S\ y) = x < y$$

Examples

Lookup function on vectors

$$vec : (N \to U) \to N \to U$$

$$vec\ B\ 0 = N_1 \qquad vec\ B\ (S\ x) = (vec\ B\ x) \times (B\ x)$$

$$lookup : (\Pi B \in N \to U)(\Pi n \in N)(\Pi x \in N)\ x < n \to vec\ B\ n \to B\ x$$

Inductive-recursive definitions

Inductive-recursive definitions can be represented by the mutual recursive definition of a labelled sum and a function

$$V : U = \hat{N} + \hat{\Pi}\ (x : V, T\ x \to V)$$

$$T : V \to U = (\hat{N} \to N,\ \hat{\Pi}\ x\ f \to (\Pi y \in T\ x) T\ (f\ y))$$

This corresponds to the equations

$$T\ \hat{N} = N$$

$$T\ (\hat{\Pi}\ x\ f) = (\Pi y \in T\ x) T\ (f\ y)$$

Type-checking

$$\frac{\Gamma \vdash T \qquad\qquad \Gamma, \vec{x} : T \vdash \vec{M} : T}{\Gamma \vdash D}$$

where D is $\vec{x} : T = \vec{M}$

Normal forms

$$
\begin{aligned}
v &::= k \mid c\ \vec{v} \mid (\lambda x.M)\sigma \mid B\sigma \mid L\sigma \\
\sigma &::= () \mid (\sigma, x = v) \mid D\sigma \\
k &::= x \mid B\sigma\ k \mid k\ v
\end{aligned}
$$

Normal forms

We use *closures* to represent infinite objects (cf. streams in scheme, Friedman and Wise)

$$(\lambda x.M)\sigma \qquad B\sigma \qquad L\sigma$$

have both a *static* part and a *dynamic* part with actual values to parameters $v_1, \ldots, v_l$

$(\lambda x.M)\sigma$ can be written $f(\vec{v})$ with defining equation $f(\vec{v})\ u = M(\sigma, x = u)$

$B\sigma$ can be written $f(\vec{v})$ with defining equations $f(\vec{v})\ (c_i\ \vec{u}) = N_i(\sigma, \vec{x_i} = \vec{u})$

$L\sigma$ can be written $d(\vec{v})$

Normal forms

$$v \;::=\; k \mid c\ \vec{v} \mid f(\vec{v}) \mid d(\vec{v})$$
$$k \;::=\; x \mid f(\vec{v})\ k \mid k\ v$$

For Gödel system T

$$f(v_1, v_2)\ 0 = v_1 \qquad\qquad f(v_1, v_2)\ (S\ n) = v_2\ n\ (f(v_1, v_2)\ n)$$

Representation of mathematical reasoning

$$M, A \ ::= \ x \mid M\ M \mid \lambda x.M \mid M\ D \mid c\ \vec{M} \mid B \mid L$$

recursive definitions for reasoning by induction

reasoning by case analysis

auxiliary lemma and definitions

Example

```
filter : {A : Set}  -> (A -> Bool) -> List A -> List A
filter p [] = []
filter p (x :: xs) with p x
...                   | true = x : filter p xs
...                   | false = filter p xs

subset  : {A : Set}  -> (p : A -> Bool) ->
          (xs : List A) -> subseteq (filter p xs) xs
subset p [] = stop
subset p (x :: xs) with p x
...                   | true = keep (subset p xs)
...                   | false = drop (subset p xs)
```

Total functional programming

"The driving force of functional programming is to make programming more closely related to mathematics. A program in a functional language ... consists of equations which are both computation rules and a basis for simple algebraic reasoning. The existing model of functional programming, although elegant and powerful, is compromised to a greater extent than is commonly recognized by the presence of partial functions. We consider a simple discipline of *total functional programming* designed to exclude the possibility of non termination."

D.A.Turner, 2004, J.U.C.S

Denotational semantics

Basic types are represented by labelled sums

Can be infinite objects denotationally

No fixed sets of primitive types

For a *strict* semantics, if the semantics of a term is $\neq\perp$ then this term is *strongly normalizable* (U. Berger, A. Spiwack, T.C.)

Total functional programming

Type theory can then be represented as the *total* subset of this language

Convertibility is decidable on normalizable terms, hence on any total fragment

Programs and proofs are terminating functional programs

This provides a way to actually program the type theoretic computations (cf. work of B. Grégoire and X. Leroy)