Computability in Europe 2006

Functional Concepts in C++

Rose H. Abdul Rauf, Anton Setzer, Ulrich Berger

Swansea

Goal:

Integrating Functional and Object-Oriented Programming

A first step:

Implementing the simply typed λ-calculus in C++.

Novelty:

Formal correctness proof.

Extensions

- recursion,
- Lazy evaluation
- λ-terms with side effects.

The simply typed λ-calculus with arithmetic functions

$$\Gamma, x : A \vdash x : A \qquad \Gamma \vdash n : \mathrm{nat}$$

$$\frac{\Gamma, x : A \vdash r : B}{\Gamma \vdash \lambda x^A r : A \to B} \qquad \frac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash r\ s : B}$$

$$\frac{\Gamma \vdash r_1 : \mathrm{nat} \ldots \Gamma \vdash r_k : \mathrm{nat}}{\Gamma \vdash f[r_1, \ldots, r_k] : \mathrm{nat}}$$

(f a name for k-ary arithmetic function $[\![f]\!]$)

Implementing types

The base type nat is implemented by the native C++ type `int`.

If the types A, B are implemented by C++ types `A`, `B`, then the type $A \to B$ is implemented by the C++ type `CA_BD` defined as follows:

```
class CA_BD_aux
{ public : virtual B operator() (A x) = 0; };

typedef CA_BD_aux*  CA_BD;
```

Implementing λ-terms

Example: $\lambda x^{\mathrm{nat}}.f\ (f\ x)$ with free variable $f^{\mathrm{nat}\to\mathrm{nat}}$.

```
class lambda1 : public Cint_intD_aux{
 public :Cint_intD f;
 lambda1( Cint_intD f)  { this-> f = f;};
 virtual int operator () (int x)
 { return (*(f))((*(f))(x)); }; };
```

Abstracting f: $\lambda f^{\mathrm{nat}\to\mathrm{nat}}\lambda x^{\mathrm{nat}}.f(fx)$

```
class lambda0 : public CCint_intD_Cint_intDD_aux{
 public :
 lambda0( ) { };
 virtual Cint_intD operator () (Cint_intD f)
 { return new  lambda1( f); } };
```

Implementing λ-terms (ctd.)

$\lambda x^{\mathrm{nat}}.2 + x$

```
class lambda2 : public Cint_intD_aux{
public :
lambda2( ) { };
virtual int operator () (int x)
{ return x + 2; };
};
```

$t = (\lambda f^{\mathrm{nat}\to\mathrm{nat}} \lambda x^{\mathrm{nat}}.f\ (f\ x))\ (\lambda x^{\mathrm{nat}}.x + 2)\ 3$

```
int t = (*((*( new lambda0( )))( new lambda2( ))))(3);
```

Proof of Correctness

Our goal is to formally prove:

If a closed term t^{nat} has value n, then our implementation evaluates t to n.

This requires:

- A formal definition of the value of a term.
- A formal model of our C++ implementation.

What is the value of a λ-term?

Operational semantics: Normal form w.r.t.

β-reduction: $(\lambda x.r)s \longrightarrow r[s/x]$

f-reduction: $f[n_1, \ldots, n_k] \longrightarrow [\![f]\!](n_1, \ldots, n_k)$

Denotational semantics: $[\![r]\!] \in \mathrm{D}$ where D is a domain of higher type functionals.

Luckily, at base type the two semantics coincide.

We will use the denotational semantics and "logical relations" to prove correctness.

Naïve denotational semantics

$\mathbf{N} = \{0, 1, 2, \ldots\}$, $X \to Y = \{f \mid f : X \to Y\}$

$$\mathrm{D}(\mathrm{nat}) = \mathbf{N}, \quad \mathrm{D}(A \to B) = \mathrm{D}(A) \to \mathrm{D}(B), \quad \mathrm{D} = \bigcup_{A \in \mathrm{Type}} \mathrm{D}(A)$$

Functional environment: $\xi : \mathrm{Var} \to \mathrm{D}$,
$\xi : \Gamma$ means $\forall x \in \mathrm{dom}(\Gamma).\xi(x) \in \mathrm{D}(\Gamma(x))$.

For $\Gamma \vdash r : A$ and $\xi : \Gamma$, we define $[\![r]\!]\xi \in \mathrm{D}(A)$:

$$[\![x]\!]\xi = \xi(x), \; [\![n]\!]\xi = n$$

$$[\![r\ s]\!]\xi = [\![r]\!]\xi([\![s]\!]\xi)$$

$$[\![\lambda x^A.r]\!]\xi(a) = [\![r]\!]\xi[x \mapsto a]$$

$$[\![f[\vec{r}]]\!] = [\![f]\!]([\![\vec{r}]\!]\xi)$$

The state monad

$$\mathrm{M}_X(Y) := X \to Y \times X$$

$$\begin{aligned}
&\mathrm{do}\{y_1 \leftarrow e_1 \;;\; y_2 \leftarrow e_2(y_1) \;;\; e_3(y_1, y_2)\}\; x = \mathsf{g} \\
&\qquad \text{let } \{(y_1, x_1) = e_1\; x, (y_2, x_2) = e_2(y_1)\; x_1\} \\
&\qquad \text{in } e_3(y_1, y_2)\; x_2
\end{aligned}$$

$$\mathrm{return} : Y \to \mathrm{M}_X(Y), \quad \mathrm{return}\; y\; x = (y, x)$$

$$\mathrm{mapM} : (Z \to \mathrm{M}_X(Y)) \to Z^* \to \mathrm{M}_X(Y^*)$$

$$\mathrm{mapM}\; f\; \vec{a} = \mathrm{do}\{y_1 \leftarrow f\; a_1 \;;\; \ldots \;;\; \mathrm{return}\; (y_1, \ldots)\}$$

$$\mathrm{get} : \mathrm{M}_X(X), \quad \mathrm{get}\; x = (x, x)$$

$$\mathrm{put} : X \to \mathrm{M}_X(\{*\}), \quad \mathrm{put}\; x\; x' = (*, x)$$

Modelling the implementation

$$
\begin{array}{lcl}
\mathrm{Addr} & = & \text{a set addresses of classes on the heap} \\
\mathrm{Constr} & = & \text{a set of constructors, i.e. class names} \\
\mathrm{Val} & = & \mathbf{N} + \mathrm{Addr} \\
\mathrm{F} & = & \text{a set of names for arithmetic C++ functions} \\
\mathrm{App} & = & \mathbf{N} + \mathrm{Var} + \mathrm{F} \times \mathrm{App}^* + \mathrm{App} \times \mathrm{App} + \mathrm{Constr} \times \mathrm{App}^* \\
\mathrm{Class} & = & \mathrm{Context} \times \mathrm{Var} \times \mathrm{Type} \times \mathrm{App} \\
\mathrm{VEnv} & = & \mathrm{Var} \rightarrow_{\mathrm{fin}} \mathrm{Val} \\
\mathrm{Heap} & = & \mathrm{Addr} \rightarrow_{\mathrm{fin}} \mathrm{Constr} \times \mathrm{Val}^* \\
\mathrm{CEnv} & = & \mathrm{Constr} \rightarrow_{\mathrm{fin}} \mathrm{Class}
\end{array}
$$

Parsing (Implementing λ-terms)

$$\mathrm{P}: \quad \mathrm{Context} \to \mathrm{Term} \to \mathrm{M}_{\mathrm{CEnv}}(\mathrm{App})$$

Evaluating λ-terms

$$\begin{aligned} \mathrm{eval}: &\quad \mathrm{CEnv} \to \mathrm{VEnv} \to \mathrm{App} \to \mathrm{M}_{\mathrm{Heap}}(\mathrm{Val}) \\ \mathrm{apply}: &\quad \mathrm{CEnv} \to \mathrm{Val} \to \mathrm{Val} \to \mathrm{M}_{\mathrm{Heap}}(\mathrm{Val}) \end{aligned}$$

$$\mathrm{P} : \quad \mathrm{Context} \to \mathrm{Term} \to \mathrm{M}_{\mathrm{CEnv}}(\mathrm{App})$$

$$\begin{aligned}
\mathrm{P}\ \Gamma\ u &= \mathrm{return}\ u,\ \text{if } u \text{ is a numeral or a variable} \\
\mathrm{P}\ \Gamma\ f[\vec{r}] &= \mathrm{do}\{\vec{a} \leftarrow \mathrm{mapM}\ (\mathrm{P}\ \Gamma)\ \vec{r}\ ;\ \mathrm{return}\ f[\vec{a}]\} \\
\mathrm{P}\ \Gamma\ (r\ s) &= \mathrm{do}\{(a,b) \leftarrow \mathrm{mapM}\ (\mathrm{P}\ \Gamma)\ (r,s)\ ;\ \mathrm{return}\ (a\ b)\} \\
\mathrm{P}\ \Gamma\ (\lambda x^A.r) &= \mathrm{do}\{a \leftarrow \mathrm{P}\ \Gamma[x \mapsto A]\ r\ ; \\
&\qquad C \leftarrow \mathrm{get}\ ;\ \mathrm{let}\ \ c = \mathrm{fresh}(C)\ ; \\
&\qquad \mathrm{put}(C[c \mapsto (\Gamma; x : A; a)])\ ; \\
&\qquad \mathrm{return}(c[\mathrm{dom}(\Gamma)])\}
\end{aligned}$$

$$\text{eval} : \quad \text{CEnv} \to \text{VEnv} \to \text{App} \to \text{M}_{\text{Heap}}(\text{Val})$$

$$\begin{aligned}
\text{eval}\ C\ \eta\ n &= \text{return}\ n \\
\text{eval}\ C\ \eta\ x &= \text{return}\ (\eta\ x) \\
\text{eval}\ C\ \eta\ f[\vec{a}] &= \text{do}\{\vec{n} \leftarrow \text{mapM}\ (\text{eval}\ C\ \eta)\ \vec{a}\ ; \\
&\qquad \text{return}\ [\![f]\!](\vec{n})\} \\
\text{eval}\ C\ \eta\ (a\ b) &= \text{do}\{(v, w) \leftarrow \text{mapM}(\text{eval}\ C\ \eta)\ (a, b)\ ; \\
&\qquad \text{apply}\ C\ v\ w\} \\
\text{eval}\ C\ \eta\ c[\vec{a}] &= \text{do}\{\vec{v} \leftarrow \text{mapM}\ (\text{eval}\ C\ \eta)\ \vec{a}\ ; \\
&\qquad H \leftarrow \text{get}\ ;\ \text{let}\ h = \text{fresh}(H)\ ; \\
&\qquad \text{put}(H[h \mapsto (c, \vec{v})])\ ;\ \text{return}(h)\}
\end{aligned}$$

$$\text{apply} : \quad \text{CEnv} \to \text{Val} \to \text{Val} \to \text{M}_{\text{Heap}}(\text{Val})$$

$$\begin{aligned}
\text{apply } C\ h\ v \quad = \quad & \text{do}\{H \leftarrow \text{get} \ ; \\
& \quad \text{let } (c, \vec{w}) = H\ h \ ; \\
& \quad \text{let } (\vec{y} : \vec{B}; x : A; a) = C\ c \ ; \\
& \quad \text{eval } C\ [\vec{y}, x \mapsto \vec{w}, v]\ a\}
\end{aligned}$$

Linking C++ with higher type functionals

The "Kripke logical relation"

$$\sim_A^C \quad \subseteq \quad (\mathrm{Val} \times \mathrm{Heap}) \times \mathrm{D}(A),$$

is defined by recursion on A as follows:

$$\begin{aligned}
(v, H) \sim_{\mathrm{nat}}^C n \quad &:\Leftrightarrow \quad v = n \\
(v, H) \sim_{A \to B}^C f \quad &:\Leftrightarrow \quad \forall C' \supseteq C, H' \supseteq H, (w, d) \in \mathrm{Val} \times \mathrm{D}(A) : \\
&\qquad (w, H') \sim_A^{C'} d \Rightarrow \mathrm{apply}\ C'\ v\ w\ H' \sim_B^{C'} f(d)
\end{aligned}$$

$$(\eta, H) \sim_\Gamma^C \xi \quad :\Leftrightarrow \quad \forall x \in \mathrm{dom}(\Gamma)(\eta(x), H) \sim_{\Gamma(x)}^C \xi(x)$$

Adequacy Theorem

Assume $\eta : \mathrm{VEnv}$, $\xi : \mathrm{FEnv}$, $\Gamma \vdash r : A$. Assume $\xi : \Gamma$, $\mathrm{P}\ \Gamma\ r\ C = (a, C')$, $C' \subseteq C''$, $(\eta, H) \sim_{\Gamma}^{C''} \xi$, and $H \subseteq H'$.

Then $\mathrm{eval}\ C''\ \eta\ a\ H' \sim_{A}^{C''} [\![r]\!]\xi$.

Proof. Induction on the typing judgement $\Gamma \vdash r : A$.

Correctness of the implementation

Assume $\vdash r : \mathrm{nat}$, $\mathrm{P}\ \emptyset\ r\ C = (a, C')$ and $C' \subseteq C''$.

Then for any heap H and environment η we have $\mathrm{eval}\ C''\ \eta\ a\ H = ([\![r]\!], H')$ for some $H' \supseteq H$.

Further work

- Study the C++ implementation of recursion, lazy evaluation, and λ-terms with side effects more systematically and prove correctness.
- Construct a less ad-hoc model of a suitable fragment of C++ (similar to Featherweight Java [IPW99]).
- Give a "monadic proof" of the Adequacy Theorem.
- Study related work, e.g.:
  - Läufer [Läu95] (1995)
  - Kiselyov [Kis98] (1998)
  - B. McNamara and Y. Smaragdakis [MS00] (2000)
  - Schupp [Sch00] (2000)

Conclusion

- The implementation of functional concepts in object-oriented (imperative) languages is feasible.
- Denotational semantics and logical relations are very useful tools for proving correctness of the implementation.

References

[ASS85] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and interpretation of computer programs*. MIT Press, 1985.

[IPW99] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[Kel97] R. M. Keller. The Polya C++ Library. Version 2.0.

Available via
`http://www.cs.hmc.edu/ keller/Polya/`,
1997.

[Kis98] O. Kiselyov. Functional style in C++: Closures, late binding, and lambda abstractions. In *ICFP '98: Proceedings of the third ACM SIGPLAN International conference on Functional programming*, page 337, New York, NY, USA, 1998. ACM Press.

[Läu95] K. Läufer. A framework for higher-order functions in C++. In *COOTS*, 1995.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(3):55–2, 1991.

[MS00] B. McNamara and Y. Smaragdakis. Functional programming in C++. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 118–129, New York, NY, USA, 2000. ACM Press.

[Plo77] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[Plo80] G. D. Plotkin. Lambda definability in the full type hierarchy. In R. Hindley and J. Seldin, editors, *To H.B. Curry: Essays in Combinatory Logic, lambda calculus and Formalisms*, pages 363 – 373. Academic Press, 1980.

[Pol81] W. Polak. Program verification based on denotation semantics. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 149–158, New York, NY, USA, 1981. ACM Press.

[Sch00] S. Schupp. Lazy lists in C++. *SIGPLAN Not.*, 35(6):47–54, 2000.

[Set03] A. Setzer. Java as a functional programming language. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs: International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002. Selected Papers.*, pages 279 – 298. LNCS 2646, 2003.