A direct translation of the Simply Typed Lambda Calculus into C++-templates

Markus Michelbrink

Department of Computer Science
University of Wales, Swansea

Joint work with Magne Haveraaen, Bergen

TYPES 2006
Nottingham, April 2006

Basic ideas

One ground type:

$$\star \approx class$$

```
template < class X_s >
struct someClass {...;};
```

is of type $\star \rightarrow \star$,

```
template < template < class X_s > class X_nss >
struct someClass {...;};
```

is of type $(\star \rightarrow \star) \rightarrow \star$,

```
template < template < class X_s > class X_nss, class X_s >
struct someClass {...;};
```

is of type $(\star \rightarrow \star) \rightarrow \star \rightarrow \star$, ...

Basic ideas

η-expand and abstract from free variables:

$$v_s : \star \quad \mapsto \quad \lambda v_s . v_s : \star \to \star$$

$$v_{nss} : \star \to \star \quad \mapsto \quad \lambda v_{nss}, v_s . v_{nss} v_s : (\star \to \star) \to \star \to \star$$

The variable $v_s : \star$ translates to

```
template < class X_s >
struct V_s {typedef typename X_s::result result;};
```

The variable $v_{nss} \sim_{\beta\eta} \lambda v_s . v_{nss} v_s : \star \to \star$ translates to

```
template < template < class X_s > class X_nss, class X_s >
struct V_nss {typedef typename X_nss<X_s>::result result;};
```

Basic ideas

Application is instantiation: The term $v_{nss}v_s : \star$ translates to

```
template < template < class X_s > class X_nss, class X_s >
struct V_nssVs {typedef typename V_nss<X_nss,V_s<X_s>>::result result;};
```

If $v \in FV(t)$: abstraction does not change anything.
E.g. $\lambda v_s.v_{nss}v_s : \star \longrightarrow \star$ translates to the template above.

If $v \in FV(t)$: add a further parameter.
E.g. $\lambda w_s.v_{nss}v_s : \star$ translates to

```
template < class Y_s, template < class X_s > class X_nss, class X_s >
struct V_nssVs {typedef typename V_nss<X_nss,V_s<X_s>>::result result;};
```

Basic ideas

Constants of ground type are given by

```
struct constant_int {typedef int result;};
struct constant_double {typedef double result;};
```

Theorem: "The translation of every closed term of ground type evaluates to the value of the closed term."

λ into λ

- $\mathsf{T}(v_i : A_1 \ldots A_m \to \star) = \lambda v_{i+1}, \ldots, v_{i+m}.v_i v_{i+1} \ldots v_{i+m}$
- $\mathsf{T}(t_0) = \lambda v_1 \ldots v_m.t_0'$ then

$$\mathsf{T}(t_0 t_1) = \underbrace{\lambda v_2 \ldots v_m.t_0'}_{\text{some } \alpha\text{-conversion}} \quad [v_1 := \mathsf{T}(t_1)]$$

- $\mathsf{T}(\lambda v.t) = \lambda v.\mathsf{T}(t) \quad for\ v \in FV(t)$
- $\mathsf{T}(\lambda v.t) = \lambda v_{new}.\mathsf{T}(t) \quad for\ v \notin FV(t)$

where $A_1 \ldots A_m \to \star := A_1 \to (A_2 \ldots (A_m \to \star) \ldots)$.

Properties

- T is welldefined on welltyped terms i.e.

$$t : A \to B \Rightarrow \mathsf{T}(t) = \lambda v : A.t'$$

  for some t'.
- $\mathsf{T}(t)$ is welltyped and

$$\mathsf{Type}(\mathsf{T}(t)) = \mathsf{Type}(t)$$

- $FV(\mathsf{T}(t)) = FV(t)$
- $T(t) \sim_{\alpha\beta\eta} t$

Adding constants of ground type $c_0, c_1, \ldots : \star$ we get

$$\mathsf{T}(t) \sim_{\beta} t$$

for every closed term $t : \star$.

What it is ...

- In terms of the lambda cube we have $(\Box, \Box)$ but without $(\star, \star)$!
  Where $\star \approx class$.
- Do we have more?

What it is ...

- We can build

```
template < int i >
struct someClass {enum ...};
```

  $\approx (i : int) \rightarrow class$. Dependent types $(\star, \Box)$?

- Function templates are build similar to class templates:

```
template < int i >
int justTheIdentity() {return i;};
```

  $\approx int \rightarrow int$. Simple Types $(\star, \star)$?

```
template < class X >
X someFunction() {...};
```

  $\approx (X : class) \rightarrow X$. Polymorphism $(\Box, \star)$?

- No, we can not use them as parameters!