Normalization in the Simply-Typed Lambda Calculus

David Walker
Department of Computer Science
Princeton University

February 6, 2002

This Lecture

- The Simply Typed Lambda Calculus
  - A normalization theorem
  - Logical relations arguments.

The Magic of Types

There are many benefits to a type system....

- Types catch stupid errors
  - using a number (17) as a function
  - dereferencing memory out of bounds
  - calling a function without enough (or the wrong type) arguments
- believe it or not, eliminating these errors goes a very long way towards increasing the security of your system
- The majority (greater than 50%) of security flaws are a result of buffer over runs (dereferencing memory out of bounds) and library or system calls with the wrong type of argument!

More Abstract Types

- Abstract types provide a way to define richer invariants and abstractions.

```
sig
  type 'a sorted_list

  val sort : 'a list -> 'a sorted_list
  val lookup : 'a -> 'a sorted_list -> bool
  val delete : 'a -> 'a sorted_list
                  -> 'a sorted_list option

end
```

More on Abstract Types

- An example more pertinent to this class:

```
sig
  type secure_file
  type open_file

  val fopen  : string -> secure_file option
  val fopen  : string -> secure_file -> open_file option
  val getc   : open_file -> char option
  val fclose : open_file -> unit
  val create : string -> string -> secure_file
end
```

- The main point: types can enforce a lot more properties than you might think at first.
- On the other hand, standard type systems, even those as powerful as ML's can't capture many important properties.
  - We can't guarantee that the programmer remembers to close all their files. (We need a *linear type system* for that.)
  - We can't guarantee that the programmer opens a limited number of files. (We need a *dependent type system* for that.)
- That's is where the research is.

Normalization

- We say that a language is *strongly normalizing* if every term in the language eventually terminates.
- Most languages are not strongly normalizing.
- The untyped lambda calculus is not strongly normalizing.

$$\Omega = (\lambda x.x\ x)(\lambda x.x\ x)$$

$$\Omega \longmapsto \Omega \longmapsto \Omega \longmapsto^{*} \Omega$$

- Of course, there are lots more programs that don't terminate
- Amazingly, the simply-typed lambda calculus is strongly normalizing. No matter what type we try to give to Ω, it will not type check.

Normalization for the Simply Typed Lambda Calculus

- More formally, we will be able to prove:

  Theorem 1 (Normalization) *If* $\cdot \vdash e : \tau$ *then there exists a value* v *such that* $e \longmapsto^{*} v$*.*

  To make things simple, we will assume there is only a single base type *unit* and a single constant (). Our proof techniques will scale to handle somewhat more sophisticated languages.

  How should we start our proof?

First Attempt

Proof by induction on the structure of e. **Wrong**. Find the mistake below.

Case x: By inversion, $x \in \mathsf{Dom}(\cdot)$, but this is a contradiction so this case cannot apply.

Case c: A constant is already a value so we are done. ($c \longmapsto^* c$) by the reflexivity rule.

Case $\lambda x : \tau_1.e$: A lambda term is also a value, so we are done.

Application Case

Case $e_1\ e_2$:
First, we need to prove a couple of little lemmas:

Lemma 1 (Congruence$*$)

- *(cong1$*$) If* $e_1 \longmapsto^* v_1$ *then* $e_1\ e_2 \longmapsto^* v_1\ e_2$*.*
- *(cong2$*$) If* $e_2 \longmapsto^* v_2$ *then* $v_1\ e_2 \longmapsto^* v_1\ v_2$*.*

Lemma 2 (Transitivity) *If* $e_1 \longmapsto^* e_2$ *and* $e_2 \longmapsto^* e_3$ *then* $e_1 \longmapsto^* e_3$*.*

Now, the proof:

1)	$e_1 \longmapsto^* v_1$	By induction
2)	$e_1\ e_2 \longmapsto^* v_1\ e_2$	By *cong*1$*$
3)	$e_2 \longmapsto^* v_2$	By induction
4)	$v_1\ e_2 \longmapsto^* v_1\ v_2$	By *cong*2$*$
5)	$\cdot \vdash v_1\ v_2 : \tau$	By type preservation
6)	$v_1 = \lambda x{:}\tau_1.e$	By inversion and canonical forms.
7)	$v_1\ v_2 \longmapsto e[v_2/x]$	By $\beta \longmapsto$
8)	$e[v_2/x] \longmapsto^* v_3$	By induction
9)	$e_1\ e_2 \longmapsto^* v_3$	By 2,4,7,8 and Transitivity

On closer inspection

Case $e_1\ e_2$

1)	$e_1 \longmapsto^* v_1$	By induction
2)	$e_1\ e_2 \longmapsto^* v_1\ e_2$	By $cong1*$
3)	$e_2 \longmapsto^* v_2$	By induction
4)	$v_1\ e_2 \longmapsto^* v_1\ v_2$	By $cong2*$
5)	$\cdot \vdash v_1\ v_2 : \tau$	By type preservation
6)	$v_1 = \lambda x{:}\tau_1.e$	By inversion and canonical forms.
7)	$v_1\ v_2 \longmapsto e[v_2/x]$	By $\beta \longmapsto$
8)	$e[v_2/x] \longmapsto^* v_3$	By induction **IH does not apply**
9)	$e_1\ e_2 \longmapsto^* v_3$	By 2,4,7,8 and Transitivity

- In step 8, in order for the induction hypothesis to apply, $e[v_2/x]$ must be strictly smaller than $e_1\ e_2$
- In general, after substitution a term may actually grow. For example:

$$(\lambda x{:}unit \to unit.x(x())) \ (\lambda z{:}unit.(\lambda y{:}unit.y)()) \longmapsto \\ (\lambda z{:}unit.(\lambda y{:}unit.y)())((\lambda z{:}unit.(\lambda y{:}unit.y)())())$$

- The initial term has depth 5 and it steps to a term with depth 6.
- Question: what is the smallest term that will grow in size after a single reduction step is taken?
- Bottom line: we can't do this proof by induction on the size of the expression.

A Logical Relation

If we aren't going to base our induction on the size of the term, what else is there?

In 1967, Tait realized that we could define a predicate (a unary relation) on terms that is indexed by the *type* of the term. Unlike the size predicate, our relation will be preserved at each step in evaluation of an expression.

Definition 3 **($\mathcal{R}_\tau(e)$)**

- $\mathcal{R}_{unit}(e)$ *iff* e *halts and* $\cdot \vdash e : unit$.
- $\mathcal{R}_{\tau_1 \to \tau_2}(e)$ *iff* e *halts and* $\cdot \vdash e : \tau_1 \to \tau_2$ *and for all terms* e' *such that* $\mathcal{R}_{\tau_1}(e')$, *we have* $\mathcal{R}_{\tau_2}(e\ e')$.

We called this sort of definition a *logical relation* because we define the property we want at base type and then we extend the definition "logically" to higher type by requiring that functions in the relation take arguments in the relation to results in the relation.
Now, we must show the following things:

1. All terms e such that $\cdot \vdash e : \tau$ actually satisfy the appropriate predicate. In other words, $\mathcal{R}_\tau(e)$.
2. For any type τ, if $\mathcal{R}_\tau(e)$ then e halts.

The corollary of these two facts is that all terms halt.

Part 2

Lemma 4 *For any type τ, if $\mathcal{R}_\tau(e)$ then e halts.*

Proof:

Trivial from the definition of the logical relation. We set it up to be easy.

$\square$

Part 1 Sublemma

In order to prove part 1, we will need to show that our relation is preserved under evaluation. The fact that the relation is defined inductively on *types* is crucial.

Lemma 5 *If* $\cdot \vdash e : \tau$ *and* $e \longmapsto e'$ *then* $\mathcal{R}_\tau(e)$ *if and only if* $\mathcal{R}_\tau(e')$.

Proof:

First, we can prove

(1) e halts if and only if e' halts. Since evaluation is deterministic (which we showed in the last lecture).

Next, we can prove our main results by induction on the structure of τ.

Case $\tau = unit$. The definition at base type expands to (1). So we are done.

Part 1 – Sublemma continued

Case $\tau_1 \to \tau_2$. For higher type, we must also prove:

1. Given $\mathcal{R}_{\tau_1}(e_1)$, if $\mathcal{R}_{\tau_2}(e\ e_1)$ then $\mathcal{R}_{\tau_2}(e'\ e_1)$.
2. Given $\mathcal{R}_{\tau_1}(e_1)$, if $\mathcal{R}_{\tau_2}(e'\ e_1)$ then $\mathcal{R}_{\tau_2}(e\ e_1)$.

Part 1:

$\cdot \vdash e_1 : \tau_1$	from the definition $\mathcal{R}_{\tau_1}(e_1)$
$\cdot \vdash e : \tau_1 \to \tau_2$	by assumption
$\cdot \vdash e\ e_1 : \tau_2$	by the typing rule for application.
$e\ e_1 \longmapsto e'\ e_1$	by $(cong1 \longmapsto)$.
$\mathcal{R}_{\tau_2}(e'\ e_1)$	by IH, (τ_2 is strictly smaller than $\tau_1 \to \tau_2$)

Part 2: Similar.

$\square$

Part 1 – The Lemma (Almost)

Lemma 6 *If* $\cdot \vdash e : \tau$ *then* $\mathcal{R}_\tau(e)$.

Proof:

By induction on the typing derivation.

Consider the case for lambda abstraction:

$$\frac{x{:}\tau_1 \vdash e : \tau_2}{\cdot \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2}$$

Once again, we cannot apply our induction hypothesis. This time the problem is that the context has the wrong form. Our relation is only defined on closed terms.

$\Box$

Part 1 – Finally

Lemma 7 *If* $x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash e : \tau$ *and* $\mathcal{R}_{\tau_1}(v_1), \ldots, \mathcal{R}_{\tau_n}(v_n)$ *then* $\mathcal{R}_\tau(e[v_1/x_1] \cdots [v_n/x_n])$.

Proof:

By induction on the typing derivation.

$\square$

Summary

- We proved a deep property of the typed lambda calculus – all well-typed terms terminate.
- The "obvious" proof technique (induction on the size of the term) didn't work because size can actually increase as reduction progresses.
- We needed to use the type structure. We defined a *logical relation* that
  - *Directly implied* the property we wanted at base type.
  - *Preserved* the property we wanted at higher type.
- The structure of this definition allowed us to carry out our proof *by induction on the structure of types.*

Future

- This technique can be modified to handle more than just the simply-typed lambda calculus. Wait for future homeworks!
- The extension to the second-order polymorphic lambda calculus (System F) is quite tricky. As a project, learn the technique and teach it to the class.
- This technique can be used to prove many more properties of the lambda calculus. It can also be extended to more general relations between terms.
- Next time, we will see an application in security.