

Principles of Programming Languages / Handouts

(PhD Program in Computer Science, DI / FCT / UNL)

Luís Caires

(version of December 4, 2009)

1 Proof Systems and Inductive Definitions

The language λ_N is the lambda calculus with natural numbers. The basic operations are functional abstraction $\text{fun}(x, e)$, representing the anonymous function that maps each parameter x to e (where the identifier x occurs) and functional application $\text{app}(f, e)$. In λ_N closed expressions (expressions without free identifiers) evaluate to simple values. In this exercise, we precisely define the syntax and semantics of λ_N and prove some properties about λ_N programs.

The signature Σ_N of the λ_N language comprises the following constructors:

0/0
 $x/0, y/0, z/0$
succ/1
fun/2
app/2

The symbols x, y, z, \dots representing program identifiers are countably many, we collect them all in the set Id of variables. We will use e, f as meta-variables for terms over Σ_N . The syntax of the λ_N language is inductively defined by the following rules for the judgment **expr** e .

$$\begin{array}{c} \mathbf{expr} \ x \qquad \mathbf{expr} \ 0 \qquad \frac{\mathbf{expr} \ e}{\mathbf{expr} \ \text{succ}(e)} \\ \frac{\mathbf{expr} \ e}{\mathbf{expr} \ \text{fun}(x, e)} \qquad \frac{\mathbf{expr} \ e_1 \ \mathbf{expr} \ e_2}{\mathbf{expr} \ \text{app}(e_1, e_2)} \end{array}$$

The values (fully simplified expressions) of the λ_N language are inductively defined by the following rules for the judgment **val** e .

$$\mathbf{val} \ 0 \qquad \mathbf{val} \ x \qquad \frac{\mathbf{val} \ e}{\mathbf{val} \ \text{succ}(e)} \qquad \frac{\mathbf{expr} \ e}{\mathbf{val} \ \text{fun}(x, e)}$$

We will use v, u as meta-variables for values, that is, expressions e such that **val** v is derivable.

1. Formalize the induction principles for **expr** e and for **val** e . That is

$$(\forall e. \mathbf{expr} \ e \Rightarrow P(e)) \Leftrightarrow ??$$

2. Using induction, prove that all values are expressions:

$$\forall e. \mathbf{val} \ e \Rightarrow \mathbf{expr} \ e$$

The one-step evaluation semantics for λ_N is inductively defined by the following rules for the judgment **step** $e_1 \ e_2$.

$$\frac{\mathbf{step} \ e \ e'}{\mathbf{step} \ \text{succ}(e) \ \text{succ}(e')} \quad \frac{\mathbf{step} \ e_1 \ e'_1}{\mathbf{step} \ \text{app}(e_1, e_2) \ \text{app}(e'_1, e_2)}$$

$$\frac{\mathbf{step} \ e_2 \ e'_2}{\mathbf{step} \ \text{app}(e_1, e_2) \ \text{app}(e_1, e'_2)} \quad \frac{\mathbf{val} \ v \quad \mathbf{subst} \ e_1 \ x \ v \ f}{\mathbf{step} \ \text{app}(\text{fun}(x, e_1), v) \ f}$$

The judgment form **subst** $e \ x \ v \ f$ is intended to state that f is the result of replacing all free occurrences of the variable x in expression e by the value v .

1. Inductively specify the set of valid **subst** $e \ x \ v \ f$ judgments by means of a proof system.
2. Carefully prove the following properties of the above systems (using induction), or give a counterexample. Interpret the results.
 - (a) $\forall e. \forall v. \forall x \in Id. \forall e'. \mathbf{expr} \ e \wedge \mathbf{val} \ v \wedge \mathbf{subst} \ e \ x \ v \ e' \Rightarrow \mathbf{expr} \ e'$.
 - (b) $\forall e. \forall e'. \mathbf{expr} \ e \wedge \mathbf{step} \ e \ e' \Rightarrow \mathbf{expr} \ e'$.
 - (c) $\forall e. \mathbf{expr} \ e \Rightarrow (\exists e'. \mathbf{step} \ e \ e') \vee \mathbf{val} \ e$.
 - (d) $\forall e. \mathbf{val} \ e \Rightarrow \neg(\exists e'. \mathbf{step} \ e \ e')$.

In general, we may want to define judgments involving not only expressions of some term algebras, but also auxiliary mathematical objects, prominently finite sets, multisets, or sequences (lists). For example, we will now define by the following proof system the finite set of *free identifiers of a program*, by means of the judgement **free** $P \ S$. **free** $P \ S$ states that the set of free identifiers of P is S .

$$\mathbf{free} \ x \ \{x\} \quad \mathbf{free} \ 0 \ \{\}$$

$$\frac{\mathbf{free} \ e \ S}{\mathbf{free} \ \text{succ}(e) \ S}$$

$$\frac{\mathbf{free} \ e_1 \ S_1 \quad \mathbf{free} \ e_2 \ S_2}{\mathbf{free} \ \text{app}(e_1, e_2) \ (S_1 \cup S_2)} \quad \frac{\mathbf{free} \ e \ S}{\mathbf{free} \ \text{fun}(x, e) \ (S \setminus \{x\})}$$

1. Using induction, interpret and carefully prove the following property:

$$\forall e, e', S, S'. \mathbf{step} \ e \ e' \wedge \mathbf{free} \ e \ S \wedge \mathbf{free} \ e' \ S' \Rightarrow S' \subseteq S$$

2. Give an example of e, e', S, S' such that $\mathbf{step} \ e \ e' \wedge \mathbf{free} \ e \ S \wedge \mathbf{free} \ e' \ S'$ and $S' \neq S$.

2 Types and Type Systems

2.1 Simple Type System and Basic Safety Properties

Consider the following rendering of a simple type system for our language λ_N . First, given the signature Σ_T

$$\begin{array}{l} \mathbb{N}/0 \\ \text{arrow}/2 \end{array}$$

we define the types:

$$\text{type } \mathbb{N} \quad \frac{\text{type } t_1 \quad \text{type } t_2}{\text{type } \text{arrow}(t_1, t_2)}$$

To facilitate reading, and adopt a standard notation, we define the abbreviation $t_1 \rightarrow t_2 \triangleq \text{arrow}(t_1, t_2)$. Typing for λ_N programs is expressed by the judgment **typeof** $C \ e \ t$, stating that expression e has type t in context C . A context is a set of declarations, assigning types to identifiers. Each declaration has the form $x : t$ where x is an identifier and t is a type. We write $\text{dom}(D)$ (the domain of D) for the set of identifiers assigned types in D . Well formed contexts are given by the following rules

$$\text{ctxt } \emptyset \quad \frac{\text{ctxt } D \quad \text{type } t}{\text{ctxt } D \cup (x : t)} \quad (\text{if } x \notin \text{dom}(D))$$

Second, we define typing of λ_N programs by the following proof system:

$$\begin{array}{c} \frac{\text{ctxt } D \quad (x : t) \in D}{\text{typeof } D \ x \ t} \\ \\ \frac{\text{ctxt } D}{\text{typeof } D \ 0 \ \mathbb{N}} \quad \frac{\text{typeof } D \ e \ t' \rightarrow t \quad \text{typeof } D \ e' \ t'}{\text{typeof } D \ \text{app}(e, e') \ t} \\ \\ \frac{\text{typeof } D \ e \ \mathbb{N}}{\text{typeof } D \ \text{succ}(e) \ \mathbb{N}} \quad \frac{\text{typeof } D \cup (x : t) \ e \ t'}{\text{typeof } D \ \text{fun}(x, e) \ t \rightarrow t'} \end{array}$$

1. Using induction, prove the following property (Weakening):

$$\forall D, D', e, t. \text{typeof } D \ e \ t \wedge \text{ctxt } (D \cup D') \Rightarrow \text{typeof } (D \cup D') \ e \ t$$

2. Using induction, prove the following property (Closure of typing under substitution):

$$\forall D, e, e', f, t, t', x. \begin{array}{l} \text{typeof } D \cup (x : t') \ e \ t \wedge \\ \text{typeof } D \ e' \ t' \wedge \\ \text{subst } e \ x \ e' \ f \Rightarrow \\ \text{typeof } D \ f \ t \end{array}$$

3. Using induction, prove the following property (Preservation of typing):

$$\forall D, e, e', t. \text{typeof } D e t \wedge \text{step } e e' \Rightarrow \text{typeof } D e' t$$

You will need to use the (Closure of typing under substitution) property to handle the case of β -reduction (function call with parameter passing).

4. Using induction, prove the following property (Progress for typed programs):

$$\forall D, e, t. \text{typeof } D e t \Rightarrow (\exists e'. \text{step } e e') \vee \text{val } e$$

5. A program e is “wrong” if it is not a value, but doesn’t reduce (that is, there is no e' such that $\text{step } e e'$). In other words, e is *stuck*.

(a) Give a couple of examples of stuck programs.

(b) Using (1-4) above, formalize and prove the following general property of type systems in the context of our λ_N language: “Well-typed programs don’t go wrong”, meaning that if a program is well-typed it will never get into an error condition during evaluation. This property is also commonly referred by “type safety”.

2.2 Strong Normalization and Logical Predicates

The following exercise concerns the establishment of properties of typed programs much stronger than type safety using the logical predicate technique. In particular, we will consider the strong normalization property, that is, “typed programs always terminate”.

We say that e_1, e_2, \dots is an evaluation sequence for e_1 if for all $i > 0$ we have $\text{step } e_i e_{i+1}$. We say that e is strongly normalizable, notation $\text{SN}(e)$, if there is no evaluation sequence for e with an infinite length.

The strong normalization property for λ_N is then stated as follows:

$$\forall e, D, t. \text{typeof } D e t \Rightarrow \text{SN}(e)$$

To prove this property we need to introduce a stronger predicate than $\text{SN}(-)$, actually, the predicates $\text{Comp}_t(-)$ discussed in the lectures.

Consider the following type-indexed predicate $\text{Comp}_t(-)$ on λ_N expressions.

$$\begin{aligned} \text{Comp}_{\mathbb{N}}(e) &\triangleq \text{SN}(e) \\ \text{Comp}_{t_1 \rightarrow t_2}(e) &\triangleq \forall e'. (\text{Comp}_{t_1}(e) \Rightarrow \text{Comp}_{t_2}(\text{app}(e, e'))) \end{aligned}$$

A predicate such as $\text{Comp}_t(-)$, defined by induction (recursively defined) on the type hierarchy, is called a *logical predicate*. As we have seen in the lectures, we may check that $\text{Comp}_t(e)$ implies $\text{SN}(e)$, but also other useful properties, such as closure under application (which is automatic by the definition of $\text{Comp}_{t_1 \rightarrow t_2}(-)$).

Day One By induction on types, prove a few fundamental properties of the logical predicate $Comp_t(-)$.

1. (Computable expressions are strongly normalizing).

$$\text{for any type } t, \forall e. Comp_t(e) \Rightarrow \mathbf{SN}(e)$$

2. (Closure under evaluation).

$$\text{for any type } t, \forall e, e'. Comp_t(e) \wedge \mathbf{step} e e' \Rightarrow Comp_t(e')$$

3. (Backward Closure).

$$\text{for any type } t, (\forall e'. \mathbf{step} e e' \Rightarrow Comp_t(e')) \Rightarrow Comp_t(e)$$

Day Two We now prove the main property of the logical predicates $Comp_t(-)$, stating that for ANY expression e well-typed by t , we have $Comp_t(e)$. This will allow us to conclude that every typed term is computable, and therefore strongly normalizing, in the light of 1 above.

In fact, we will need to prove the more flexible statement, that $Comp_t(-)$ holds for any well typed expression even after such expression is subject to substitution of their free identifiers for computable expressions of the appropriate type. This generality is needed to show that $\mathbf{typeof} D \text{ fun}(x, g) t_1 \rightarrow t_2$ implies $Comp_{t_1 \rightarrow t_2}(\text{fun}(x, g))$, as we have discussed in the lectures, by analysing the evaluation behavior of $\text{fun}(x, g)$ in 1 below.

To express this more general property, it is useful to introduce a notion of “computable substitution”. Given a typing context D , a computable substitution for D is a function θ from the set of identifiers to the set of expressions that, for each $x : t \in D$, assigns to the identifier x an expression f such that $Comp_t(f)$.

For every context D , and substitution θ , we define

$$Comp_D(\theta) \triangleq \forall x : t \in D \Rightarrow Comp_t(\theta(x))$$

We write $Comp_D(\theta)$ to assert that θ is a computable substitution for D . We can now formulate the main property as follows:

$$\forall e, D, t, \theta. \mathbf{typeof} D e t \wedge Comp_D(\theta) \Rightarrow Comp_t(\theta(e))$$

1. Prove the main property by induction on the derivation of $\mathbf{typeof} D e t$. All proof cases in are very simple to handle, using the inductive hypotheses, except the case for $\text{fun}(x, g)$, where you will also need to use all the properties 1 – 2 – 3 above.
2. Check that if x is an identifier, then $Comp_t(x)$ for any type t (proof by induction on t).
3. Combining the above results, prove strong normalization of λ_N :

$$\forall e, D, t. \mathbf{typeof} D e t \Rightarrow \mathbf{SN}(e)$$

2.3 Strong Normalization with Inductive Types

In this exercise, you will state and prove type safety and strong normalization for a generic programming language λ_T with inductive data types and recursive definitions, extending the basic approach of Gödel System T.

Programming Language We start by assuming a finite set of inductive types $\alpha_1, \alpha_2, \dots, \alpha_k$. Examples of inductive types are the natural numbers, lists, trees, etc. Values of an inductive type are built by the application of data constructors. In this exercise, we will not fix a definite set of inductive types or constructors, but discuss a language with an arbitrary set of inductive types, and constructors. So, many of the definitions below are actually schematic, in order to obtain general results, applicable to any language of this form.

The types of λ_T are defined from the signature

$$\begin{array}{l} \alpha_1/0, \alpha_2, \dots \\ \text{arrow}/2 \end{array}$$

by the rules:

$$\text{type } \alpha_i \quad \frac{\text{type } t_1 \quad \text{type } t_2}{\text{type } \text{arrow}(t_1, t_2)}$$

The signature Σ_T of the λ_T language comprises the basic constructors:

$$\begin{array}{l} x/0, y/0, z/0, \dots \\ \text{fun}/2 \\ \text{app}/2 \end{array}$$

It also includes, for each inductive type α ,

- a finite set of data constructors, $C_1^\alpha, C_2^\alpha, \dots, C_{k_\alpha}^\alpha$. Each constructor is sorted, that is, comes equipped with *sort*, specifying the number of arguments, and the type of each such argument. A sort for a constructor C for the inductive type α is a tuple of the form $(t_1, \dots, t_n, \alpha^1, \dots, \alpha^p)$ where $n \geq 0, p \geq 0$ and for all t_i , we have **type** t_i and t_i does not contain α . Given a constructor C we denote its sort by $\text{sort}(C)$.
- a recursor $\text{rec}_\alpha/k_\alpha$. Notice that the arity of the recursor rec_α is the number of constructors of inductive type α .

For example, for the type \mathbb{N} of natural numbers, we would introduce two constructors: the constructor `zero` with $\text{sort}(\text{zero}) = ()$ and the constructor `succ` with $\text{sort}(\text{succ}) = (\mathbb{N})$.

For the type `Nlist` of lists of natural numbers, we would consider the constructors `nil` with $\text{sort}(\text{nil}) = ()$, and the constructor `cons` with $\text{sort}(\text{cons}) = (\mathbb{N}, \text{Nlist})$. We will also have the recursors $\text{rec}_\mathbb{N}/2$ and $\text{rec}_{\text{Nlist}}/2$.

The syntax of the λ_T language is inductively defined by the following rules.

$$\begin{array}{c}
\mathbf{expr} \ x \\
\hline
\mathbf{expr} \ e \\
\hline
\mathbf{expr} \ \mathbf{fun}(x, e) \\
\hline
\mathbf{expr} \ e_1 \ \dots \ \mathbf{expr} \ e_n \\
\hline
\mathbf{expr} \ \mathbf{C}(e_1, \dots, e_n) \\
\hline
\mathbf{expr} \ e_1 \quad \mathbf{expr} \ e_2 \\
\hline
\mathbf{expr} \ \mathbf{app}(e_1, e_2) \\
\hline
\mathbf{expr} \ e_1 \ \dots \ \mathbf{expr} \ e_{k_\alpha} \\
\hline
\mathbf{expr} \ \mathbf{rec}_\alpha(e_1, \dots, e_{k_\alpha})
\end{array}$$

Type System As in the previous section, we define the type system for λ_T by first introducing a notion of typing context.

$$\mathbf{ctxt} \ \emptyset \quad \frac{\mathbf{ctxt} \ D \quad \mathbf{type} \ t}{\mathbf{ctxt} \ D \cup (x : t)} \quad (\text{if } x \notin \text{dom}(D))$$

Typing of λ_T programs is defined by the following schematic proof system:

$$\begin{array}{c}
\mathbf{ctxt} \ D \quad (x : t) \in D \\
\hline
\mathbf{typeof} \ D \ x \ t \\
\hline
\mathbf{ctxt} \ D \quad \mathbf{sort}(\mathbf{C}_k^\alpha) = (t_1, \dots, t_n, \alpha^1, \dots, \alpha^p) \quad \mathbf{typeof} \ D \ e_i \ t_i \quad \mathbf{typeof} \ D \ f_j \ \alpha \\
\hline
\mathbf{typeof} \ D \ \mathbf{C}_k^\alpha(e_1, \dots, e_n, f_1, \dots, f_p) \ \alpha \\
\hline
\frac{\mathbf{typeof} \ D \ e \ t' \rightarrow t \quad \mathbf{typeof} \ D \ e' \ t'}{\mathbf{typeof} \ D \ \mathbf{app}(e, e') \ t} \quad \frac{\mathbf{typeof} \ D \cup (x : t) \ e \ t'}{\mathbf{typeof} \ D \ \mathbf{fun}(x, e) \ t \rightarrow t'} \\
\hline
\mathbf{typeof} \ D \cup \{x_{n_i}^i : t_{n_i}^i, y_{p_i}^i : \alpha^{ip_i}, z_{p_i}^i : t\} \ e_i \ t \quad \mathbf{sort}(\mathbf{C}_i^\alpha) = (\overline{t_{n_i}}, \overline{\alpha^{p_i}}) \quad (\text{for } i = 1 \dots k_\alpha) \\
\hline
\mathbf{typeof} \ D \ \mathbf{rec}_\alpha(\mathbf{fun}(x_{n_1}^1 y_{p_1}^1 z_{p_1}^1, e_1), \dots, \mathbf{fun}(x_{n_{k_\alpha}}^{k_\alpha} y_{p_{k_\alpha}}^{k_\alpha} z_{p_{k_\alpha}}^{k_\alpha}, e_{k_\alpha})) \ \alpha \rightarrow t
\end{array}$$

Notice that we are using the notation $\mathbf{fun}(z_1, \dots, z_p, e)$ to denote the expression $\mathbf{fun}(z_1, \mathbf{fun}(z_2, \dots, \mathbf{fun}(z_p, e)))$, and so on.

1. Define λ_T programs to:
 - Compute the multiplication of two natural numbers
 - Reverse a list of natural numbers
 - Given a list of n functions f_i of type $\mathbb{N} \rightarrow \mathbb{N}$ and a natural number K , compute $f_1(f_2(f_3 \dots f_n(K) \dots))$
2. Define the operational semantics for λ_T , using a call-by-value small step semantics.
3. State and prove the type safety property for λ_T . You are only required to state the substitution lemma, not to prove it. But prove the subject reduction property in detail.
4. State and prove the strong normalization property for λ_T , using the logical relations method (Hint: the case for proving that the recursors are computable follows the lines of the proof discussed in the lectures for the Gödel System T).