

# Type Theory (15-814) Fall 2006

## Homework 2: Types in Gödel's T

William Lovas (wlovas@cs)

Out: Wednesday, October 11, 2006

Due: Thursday, October 19, 2006 (before 1:30 pm)

For the entirety of this homework, you'll be working with Gödel's T, extended with products. In the first part of the homework, you'll learn about the expressive power of Gödel's T and prove its key properties, safety and termination. Then, in the second part, you'll build a typechecker and evaluator for T in Standard ML, allowing you to experience the power of induction first-hand, interactively!

Your solution should be submitted electronically as three files: `solution.pdf`, `typecheck.sml`, and `evaluate.sml`. Submit these files by copying them to the directory

`/afs/cs.cmu.edu/academic/class/15814/handin/<userid>/hw2`

Good luck!

### 1 Theory: Gödel's T

In class we discussed Gödel's T, a simple type theory for computing functions of natural numbers. We extend this language with product types  $\tau_1 \times \tau_2$ : the introduction form is a pair of expressions (written  $\langle e_1, e_2 \rangle$ ) and the elimination forms are the first and second projection operators (written `fst`  $e$  and `snd`  $e$ ).

$$\begin{aligned} \tau &::= \text{nat} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2 \\ e &::= \text{zero} \mid \text{succ } e \mid \text{rec}(e, e_1, x:\tau. e_2) \mid \lambda x:\tau. e \mid e_1 e_2 \mid \langle e_1, e_2 \rangle \mid \text{fst } e \mid \text{snd } e \end{aligned}$$

The complete typing rules and operational semantics are shown in Figures 1 and 2.

#### 1.1 Warm-up: Hacking Gödel's T

As proven in class, all programs written in Gödel's T terminate<sup>1</sup>. Yet we can define a wide variety of number-theoretic functions using just these few primitive constructs. Recall the definition of `plus` given in class:

$$\overline{\text{plus}} := \lambda m:\text{nat}. \lambda n:\text{nat}. \text{rec}(m, n, x:\text{nat}. \text{succ } x)$$

Intuitively, this says that given  $m$  and  $n$ , if  $m = 0$ , then  $m + n = n$ , and if  $m = \text{succ } m'$ , then  $m + n = \text{succ } (m' + n)$ .

**Exercise 1.** Define the following functions as terms in Gödel's T:

1. `times`, where  $\overline{\text{times } m\ n} = \overline{m * n}$

---

<sup>1</sup>You'll extend the proof to Gödel's T with products in Exercise 6.

2.  $\overline{\text{fact}}$ , where  $\overline{\text{fact } n} = \overline{n!}$

( $n!$  is the factorial function at  $n$ :

$$\begin{aligned} 0! &:= 1 \\ n! &:= n * (n - 1)! && (n > 0) \end{aligned}$$

**Hint:** You'll need to use pairs—remember how we defined  $\overline{\text{pred}}$  for Church numerals in the untyped  $\lambda$ -calculus.)

3.  $\overline{\text{ack}}$ , where  $\overline{\text{ack } m n} = \overline{\text{ack}(m, n)}$

( $\text{ack}(m, n)$  is the Ackermann function at  $m$  and  $n$ :

$$\begin{aligned} \text{ack}(0, n) &:= n + 1 \\ \text{ack}(m, 0) &:= \text{ack}(m - 1, 1) && (m > 0) \\ \text{ack}(m, n) &:= \text{ack}(m - 1, \text{ack}(m, n - 1)) && (m > 0, n > 0) \end{aligned}$$

**Hint:** It is somewhat surprising that this function can be defined in Gödel's T, since it is *not* primitive recursive, in the usual sense of the term! Its definition requires the use of Gödel's recursor at a function type.)

## 1.2 Type Safety

In class we sketched a proof of the *type safety* of Gödel's T. This theorem forms the cornerstone of all of our developments for the rest of the semester, and indeed throughout the field of PL research. Type safety relates the typing of expressions  $\Gamma \vdash e : \tau$  to their operational semantics  $e \mapsto e'$  in a particular way—roughly speaking, well-typed programs don't "go wrong" when evaluated.

A key lemma used in the proof of type safety is the Substitution theorem, which allows us to dispense with typing assumptions in  $\Gamma$  given an expression of the appropriate type. In class it was written as an admissible rule of inference:

$$\frac{\Gamma, x:\tau_2 \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash [e_2/x] e_1 : \tau_1}$$

It can be written equivalently as a theorem statement.

**Theorem (Substitution).** If  $\Gamma, x:\tau_2 \vdash e_1 : \tau_1$  and  $\Gamma \vdash e_2 : \tau_2$ , then  $\Gamma \vdash [e_2/x] e_1 : \tau_1$ .

**Exercise 2.** Prove the Substitution theorem for Gödel's T with products. Please show all cases.

Two other important lemmas were the Canonical Forms lemma and the Inversion lemma. Canonical Forms says that the types of values dictate their shapes: for example, one case of the Canonical Forms lemma says that if  $\cdot \vdash e : \text{nat}$  and  $e$  *value*, then either  $e = \text{zero}$  or  $e = \text{succ } e'$  for some  $e'$ ; other cases delineate the shapes of other sorts of values.

The Inversion lemma lets us apply the typing rules "in reverse"—given the conclusion of a typing rule, the premises must hold as well. For example, Inversion on natural numbers would say that if  $\Gamma \vdash \text{succ } e : \text{nat}$ , then  $\Gamma \vdash e : \text{nat}$ . In a language as simple as Gödel's T, the Inversion lemma is trivial to prove: one simply observes that the rules are *syntax-directed*—only one rule's conclusion applies to any given expression. Nonetheless, it is an important lemma to understand.

**Exercise 3.** State the cases of the Canonical Forms and Inversion lemmas having to do with products; you need not prove them. (You may find it helpful to try doing Exercise 4 first, to motivate these lemmas.)

Type safety proper is proven as the conjunction of two theorems: Progress and Preservation. Progress says that a well-typed term’s evaluation never gets stuck—well-typed programs don’t contain “illegal instructions”. Preservation says that a well-typed term can’t “run off into the weeds” and become ill-typed—evaluation preserves typing. Formally,

**Theorem (Progress).** If  $\cdot \vdash e : \tau$ , then either  $e$  value or  $e \mapsto e'$ .

**Theorem (Preservation).** If  $\cdot \vdash e : \tau$  and  $e \mapsto e'$ , then  $\cdot \vdash e' : \tau$ .

Together, these theorems tell us that well-typed programs don’t go wrong: we can always move forward from a well-typed term, and in doing so, the term remains well-typed.

**Exercise 4.** Prove the Progress and Preservation theorems for Gödel’s T with products. You may optionally omit the cases having to do with  $\rightarrow$  types, since those were covered in lecture, but you may find it instructive to reproduce them on paper.

### 1.3 Termination

Recall from our discussion in class that evaluation of (well-typed) expressions in Gödel’s T always terminates—all expressions have normal forms. This means, of course, that  $\lambda$ -terms like the Y-combinator and  $(\lambda x. x x) (\lambda x. x x)$  can’t be given a type in Gödel’s T. Intuitively, the restrictions imposed by T’s type structure rule out such potentially-looping programs, which almost always involve some form of self-application.

**Exercise 5.** Show that in Gödel’s T, there are no  $\tau$  and  $\tau'$  such that  $\Gamma \vdash (\lambda x:\tau. x x) : \tau'$ .

More formally, we can prove using induction and the method of logical relations that for any  $e$  such that  $\cdot \vdash e : \tau$ , evaluation of  $e$  reaches a normal form. Remember that the trick to proving such a theorem was to find a strong enough generalization of the induction hypothesis. We settled on defining a family of predicates  $\text{HT}_\tau(e)$  for expressions  $e : \tau$ , which said that not only does  $e$  terminate, but also that its subterms *hereditarily* terminate in some appropriate sense. For example, at function types, we said that

$\text{HT}_{\tau_1 \rightarrow \tau_2}(e)$  if and only if

1.  $e \mapsto^* \lambda x:\tau_1. e'$  for some  $e'$ , and
2. for any  $e_1 : \tau_1$ , if  $\text{HT}_{\tau_1}(e_1)$  then  $\text{HT}_{\tau_2}([e_1/x] e')$

The invariant of *hereditary termination* provided by this relation allowed us to prove that it holds for all terms in Gödel’s T.

**Theorem (Hereditary Termination).** If  $x_1:\tau_1, \dots, x_n:\tau_n \vdash e : \tau$  and  $\text{HT}_{\tau_1}(e_1), \dots, \text{HT}_{\tau_n}(e_n)$ , then  $\text{HT}_\tau([e_1, \dots, e_n/x_1, \dots, x_n] e)$ <sup>2</sup>.

Although the extension of Gödel’s T to include products seems quite straightforward, we’d like to be sure that all computations still terminate—subtle changes sometimes have deep consequences, and the only way to know for sure is to try the proof!

**Exercise 6.** Define  $\text{HT}_{\tau_1 \times \tau_2}(e)$  and prove the product cases of the Hereditary Termination theorem.

---

<sup>2</sup>(In class we wrote  $\hat{e}$  for  $[e_1, \dots, e_n/x_1, \dots, x_n] e$ .)

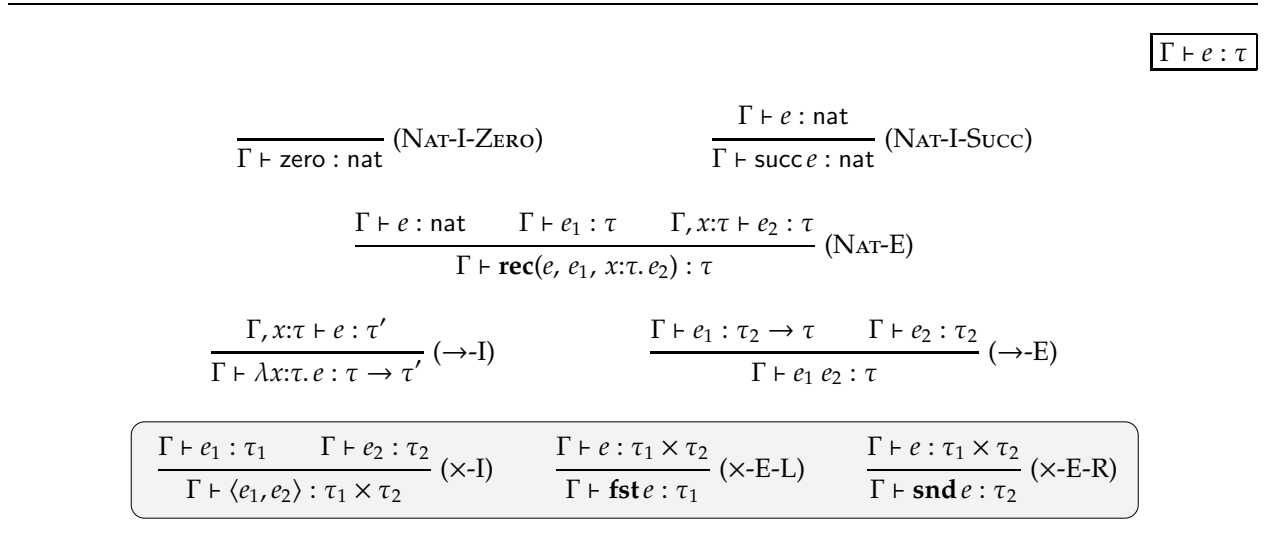


Figure 1: Typing rules for Gödel's T with products

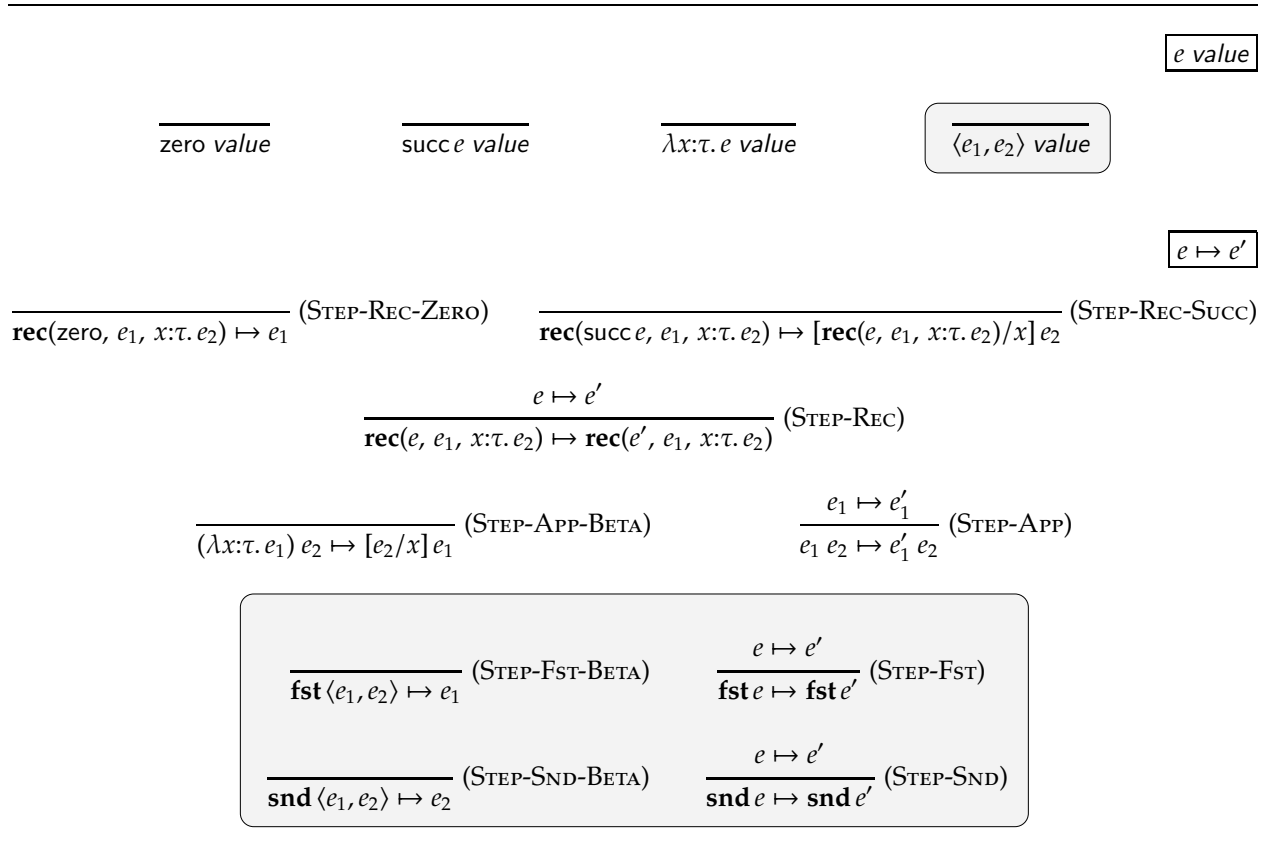


Figure 2: Call-by-name operational semantics of Gödel's T with products

## 2 Programming: A typechecker and an evaluator

Now that you're well-versed in the metatheory of Gödel's T, the logical next step is to implement it! In this section, you're asked to implement a typechecker and an evaluator for Gödel's T with products, suitable for checking your answers to Exercise 1.

You need only implement two top-level functions: `Typecheck.synthType` and `Evaluate.evaluate`. The typechecker should implement the rules for  $\Gamma \vdash e : \tau$  shown in Figure 1, and the evaluator should implement call-by-name evaluation, the reflexive, transitive closure  $\mapsto^*$  of the  $e \mapsto e'$  judgement defined in Figure 2. You can raise `Typecheck.Check ...` and `raise Evaluate.Eval ...` with appropriate error messages to signal any errors. The relevant signatures are shown in Figure 3.

The concrete syntax you'll use with your typechecker and evaluator is a variation of the one we used for the untyped  $\lambda$ -calculus:

$$\begin{array}{ll} ty ::= \text{nat} & exp ::= \text{zero} \mid \text{succ } exp \mid \text{rec}(exp, exp_1, x:ty. exp_3) \\ \mid ty_1 \rightarrow ty_2 & \mid \%x:ty. exp \mid exp_1 exp_2 \\ \mid ty_1 * ty_2 & \mid (exp_1, exp_2) \mid \text{fst } exp \mid \text{snd } exp \end{array}$$

The usual conventions apply: function application associates to the left and the scope of a binder extends as far to the right as possible, with parentheses serving to disambiguate.

In addition to the above syntax, the parser lets you use two derived forms:

1. Numeric literals like 3 will automatically be rendered internally as inductive nats like `succ (succ (succ zero))`.
2. Let-binding `let x : ty = exp1 in exp2` will automatically be rendered internally as `(%x : ty. exp2) exp1`.

The structure `Top` contains a number of useful functions for testing your programs; its signature is shown in Figure 4. In particular, `Top.process*` run your typechecker and evaluator on various forms of input, and `Top.enable*/Top.disable*` let you control the behavior of `Top.process*`:

1. `Top.{enable,disable}Typechecking` let you decide whether `Top.process*` should typecheck your inputs or not. By default, they perform typechecking, but you may wish to experiment with ill-typed terms. (When grading, though, we will only test your evaluator's behavior on well-typed terms.)
2. `Top.{enable,disable}Forcing` control the printing of values. With forcing enabled, extra transformations will be applied to your evaluator's output to convert it from a *call-by-name* value to a *call-by-value* value. This is useful for testing number-theoretic functions, for example, to reveal the value underneath a `succ` output from a function.

Forcing is enabled by default, but you can turn it off if you only want to see the call-by-name values produced by your evaluator.

As always, you're encouraged to submit any interesting test cases you devise. Since some subset of submitted test cases will be used for grading, it's in your best interests to submit cases your programs handle correctly.

---

```
signature TYPECHECK =  
  sig  
    exception Check of string  
  
    (* synthesize a type for an expression, raising Check on error *)  
    val synthType : Syntax.exp -> Syntax.ty  
  end
```

---

```
signature EVALUATE =  
  sig  
    exception Eval of string  
  
    (* compute the call-by-name value of an expression, raising Eval on error *)  
    val evaluate : Syntax.exp -> Syntax.exp  
  end
```

---

Figure 3: Signatures for the typechecker and the evaluator

---

```
signature TOP =  
  sig  
  
    (* whether to typecheck terms or not (on by default) *)  
    val enableTypechecking : unit -> unit  
    val disableTypechecking : unit -> unit  
  
    (* whether to force CBN values to CBV values (on by default) *)  
    val enableForcing : unit -> unit  
    val disableForcing : unit -> unit  
  
    (* start a top-loop *)  
    val processLoop : unit -> unit  
  
    (* typecheck and evaluate T exps in a string *)  
    val processString : string -> unit  
  
    (* typecheck and evaluate T exps from a file *)  
    val processFile : string -> unit  
  end
```

---

Figure 4: Signature of the top-level interface Top