A Programming Problem

Robert Harper
Carnegie Mellon University

IFIP WG2.8 June 2008

Problem Description

Gödel's T

Types:

$$\begin{array}{lll} \tau & ::= & \mathtt{nat} \quad \text{naturals} \\ & \mid & \tau_1 \rightarrow \tau_2 \quad \text{functions} \end{array}$$

Expressions:

$$\begin{array}{llll} e & ::= & x & \text{variable} \\ & \mid & \mathtt{z} & \text{zero} \\ & \mid & \mathtt{s}(e) & \text{successor} \\ & \mid & \mathtt{rec}[\tau](e; e_0; x.y.e_1) & \text{recursor} \\ & \mid & \lambda(x{:}\tau.\, e) & \text{lambda} \\ & \mid & e_1(e_2) & \text{application} \end{array}$$

Judgements:

$$\begin{array}{ll} \Gamma \vdash e : \tau & \text{Typing Judgement} \\ \Gamma \vdash e_1 \equiv e_2 : \tau & \text{Maximal Consistent Congruence} \end{array}$$

Definability in T

A function $F : \mathbb{N} \to \mathbb{N}$ is *definable* in **T** iff there exists a term e_F of type $\mathtt{nat} \to \mathtt{nat}$ such that $F(m) = n$ iff $e_F(\overline{m}) \equiv \overline{n}$.

Theorem 1 (Gödel)**.** *The functions definable in* ***T*** *are those provable total in* ***HA****.*

Proof. Normalization proof is formalizable in **HA**. Totality proofs in **HA** can be erased to terms in **T**. □

Using Gödel-numbering and diagonalization one may exhibit a function that is *not* definable in **T**.

An Undefinable Function

For an expression e of **T**, let $\ulcorner e \urcorner \in \mathbb{N}$ be the Gödel-number of e.

Let the function $E : \mathbb{N} \to \mathbb{N}$ be such that if e is a closed term of type $\mathtt{nat} \to \mathtt{nat}$, then $E(\ulcorner e \urcorner) = n$ iff $e(\overline{\ulcorner e \urcorner}) \equiv \overline{n}$.

Theorem 2. *The function E is not definable in* ***T****.*

Proof. Suppose e_E defines E, and let $e_D = \lambda(x\mathtt{:nat.}\,\mathtt{s}(e_E(x)))$. We have

$$e_D(\overline{\ulcorner e_D \urcorner}) \equiv \mathtt{s}(e_E(\overline{\ulcorner e_D \urcorner})) \quad (1)$$

$$\equiv \mathtt{s}(e_D(\overline{\ulcorner e_D \urcorner})). \quad (2)$$

This contradicts consistency of equivalence in **T**. $\square$

An Undefinable Function

For an expression e of **T**, let $\ulcorner e \urcorner \in \mathbb{N}$ be the Gödel-number of e.

Let the function $E : \mathbb{N} \to \mathbb{N}$ be such that if e is a closed term of type $\mathtt{nat} \to \mathtt{nat}$, then $E(\ulcorner e \urcorner) = n$ iff $e(\overline{\ulcorner e \urcorner}) \equiv \overline{n}$.

Theorem 4. *The function E is not definable in* ***T****.*

Proof. Suppose e_E defines E, and let $e_D = \lambda(x{:}\mathtt{nat}.\,\mathtt{s}(e_E(x)))$. We have

$$e_D(\overline{\ulcorner e_D \urcorner}) \equiv \mathtt{s}(e_E(\overline{\ulcorner e_D \urcorner})) \quad (1)$$

$$\equiv \mathtt{s}(e_D(\overline{\ulcorner e_D \urcorner})). \quad (2)$$

This contradicts consistency of equivalence in **T**. $\square$

Corollary 5. *The function E is not provably total in* ***HA****.*

Definability in F

Theorem 6. *The function E is provably total in **HA**$_2$.*

Proof. Essentially, can comprehend all possible computability predicates in order to account for all possible programs. □

Definability in F

Theorem 9. *The function E is provably total in **HA**$_2$.*

Proof. Essentially, can comprehend all possible computability predicates in order to account for all possible programs. □

Theorem 10 (Girard)**.** *A function on the natural numbers is definable in System **F** iff it is provably total in **HA**$_2$.*

Corollary 11. *The function E is definable in System **F**.*

Definability in F

Theorem 12. *The function E is provably total in* ***HA***$_2$.

Proof. Essentially, can comprehend all possible computability predicates in order to account for all possible programs. □

Theorem 13 (Girard)**.** *A function on the natural numbers is definable in System* ***F*** *iff it is provably total in* ***HA***$_2$.

Corollary 14. *The function E is definable in System* ***F***.

This raises an interesting programming problem . . . .

The Problem

Give an explicit definition of the function E in System **F**.

In other words, define an evaluator for Gödel's **T** in Girard's **F**.

This seems to be a hard problem!

1. The evaluator must be *manifestly total*, in accordance with Girard's Theorem.
2. The implicit proof of its totality must encompass *all possible* proofs of termination formalizable in (first-order) **HA**.

Some Guidelines

You may use any sort of term representation you'd like, as long as it's obvious that it can be Church-encoded. That is, you are permitted to use inductively defined types in **F**.

Some Guidelines

You may use any sort of term representation you'd like, as long as it's obvious that it can be Church-encoded. That is, you are permitted to use inductively defined types in **F**.

You may use a lexicographic extension of structural induction to any finite number of places. That is, may use a nested structural induction in which the outer induction dominates the inner induction.

Some Guidelines

You may use any sort of term representation you'd like, as long as it's obvious that it can be Church-encoded. That is, you are permitted to use inductively defined types in **F**.

You may use a lexicographic extension of structural induction to any finite number of places. That is, may use a nested structural induction in which the outer induction dominates the inner induction.

Any characterization of equivalence in **T** sufficient for definability of computations of type `nat` is acceptable. You need not prove that it is the maximal consistent congruence.

Partial Credit

Partial credit will be awarded for solutions to any of these problems:

1. Show that E is definable in **Agda** or **Coq**, using dependent types and large eliminations to define families of types indexed by an inductive type.
2. Show that the analogue of E for simply typed λ-calculus with Booleans is definable in System **F**.

The first may or may not be "on track" for a full-credit solution, but the second definitely is.

Sketch of Solution

Contributed Solutions

Stephanie, Koen, and Lennart contributed similar solutions to the problem as stated, using Coq, Haskell, and Agda, respectively.

- Interpret **T** types as Coq/Haskell/Agda types.
- Adequate for `nat`, and hence for defining E as specified.

These appear to be definable in **F**, if pressed.

Contributed Solutions

Stephanie, Koen, and Lennart contributed similar solutions to the problem as stated, using Coq, Haskell, and Agda, respectively.

- Interpret **T** types as Coq/Haskell/Agda types.
- Adequate for `nat`, and hence for defining E as specified.

These appear to be definable in **F**, if pressed.

Congratulations on clean solutions to the stated problem!

Contributed Solutions

Stephanie, Koen, and Lennart contributed similar solutions to the problem as stated, using Coq, Haskell, and Agda, respectively.

- Interpret **T** types as Coq/Haskell/Agda types.
- Adequate for `nat`, and hence for defining E as specified.

These appear to be definable in **F**, if pressed.

Congratulations on clean solutions to the stated problem!

But I had a little more in mind, despite what I in fact asked ...

- Compute *canonical forms* at all types (numerals at `nat`).
- Equivalence is characterized as having the same canonical form.

Solution in Agda

Represent the standard logical relations argument as a dependently typed functional program.

$$E \in \prod G : \mathit{Ctx}.\ \prod t : \mathit{Tp}.\ \prod e : \mathit{Tm}.$$
$$G \vdash e : t \longrightarrow \mathit{Comp}^*[G](\gamma) \longrightarrow \mathit{Comp}[t](\hat{\gamma}(e)).$$

Makes use of inductive definitions of types and families:

1. Syntax: *Tp*, *Tm*, *Ctx*.
2. Typing judgement $G \vdash e : t$.
3. Computability predicates: $\mathit{Comp}[t](e)$ and $\mathit{Comp}^*[G](\gamma)$.

Agda neatly and conveniently supports writing this code!

Solution in Agda

Represent the standard logical relations argument as a dependently typed functional program.

$$E \in \prod G : \mathit{Ctx}.\ \prod t : \mathit{Tp}.\ \prod e : \mathit{Tm}.$$
$$G \vdash e : t \longrightarrow \mathit{Comp}^*[G](\gamma) \longrightarrow \mathit{Comp}[t](\hat{\gamma}(e)).$$

Makes use of inductive definitions of types and families:

1. Syntax: *Tp*, *Tm*, *Ctx*.
2. Typing judgement $G \vdash e : t$.
3. Computability predicates: $\mathit{Comp}[t](e)$ and $\mathit{Comp}^*[G](\gamma)$.

Agda neatly and conveniently supports writing this code!

But it's not in System **F**, nor is it obvious how to transform it into System **F**.

Solution in F

A strategy that works:

1. Translate **T** into $\mathbf{T}_\omega$, an *infinitary* version of **T**.
2. Define E_ω for $\mathbf{T}_\omega$.
3. Obtain E by composing E_ω with translation.

Why this helps:

1. Translation takes care of the termination proofs once and for all so that the evaluator need not be concerned with them.
2. It is easy to define conversion to canonical forms for $\mathbf{T}_\omega$ (no harder than for Booleans).
3. Key Lemma: *structural cut elimination*, aka *hereditary substitution*.

Infinitary T

The ω-rule for arithmetic as an alternative to induction:

$$\frac{A(0)\ \mathsf{true} \quad A(1)\ \mathsf{true} \quad A(2)\ \mathsf{true} \quad \ldots}{x \in \mathsf{nat} \vdash A(x)\ \mathsf{true}}$$

Premise is an infinite sequence of proofs.

Define **T**$_\omega$ similarly:

$$\frac{\phi(0) : \tau \quad \phi(1) : \tau \quad \phi(2) : \tau \quad \ldots}{x : \mathtt{nat} \vdash \mathsf{case}\ x\ \mathsf{of}\ \phi : \tau}$$

Here ϕ is an infinite sequence of terms and we have

$$\mathsf{case}\ \overline{n}\ \mathsf{of}\ \phi \equiv \phi(n) \quad (n \in \mathbb{N})$$

Infinitary T

The ω-rule for arithmetic as an alternative to induction:

$$\frac{A(0)\ \text{true} \quad A(1)\ \text{true} \quad A(2)\ \text{true} \quad \ldots}{x \in \text{nat} \vdash A(x)\ \text{true}}$$

Premise is an infinite sequence of proofs.

Define $\mathbf{T}_\omega$ similarly:

$$\frac{\phi(0) : \tau \quad \phi(1) : \tau \quad \phi(2) : \tau \quad \ldots}{x : \mathtt{nat} \vdash \text{case}\, x\, \text{of}\, \phi : \tau}$$

Here ϕ is an infinite sequence of terms and we have

$$\text{case}\, \overline{n}\, \text{of}\, \phi \equiv \phi(n) \quad (n \in \mathbb{N})$$

But wait! What sort of thing is ϕ?

It is a *meta-function* in the ambient meta-theory . . . .

Translating T to T$_\omega$

Replace all occurrences of $\texttt{rec}[\tau](e; e_0; x\,.\,y\,.\,e_1)$ by case e of ϕ, where

$$\begin{aligned} \phi(0) &= e_0^* \\ \phi(n+1) &= \text{let } x \text{ be } \overline{n} \text{ and } y \text{ be } \phi(n) \text{ in } e_1^* \end{aligned}$$

The meta-function ϕ is defined by primitive recursion.

Translating T to $\mathbf{T}_\omega$

Replace all occurrences of $\texttt{rec}[\tau](e; e_0; x\,.\,y\,.\,e_1)$ by case e of ϕ, where

$$\begin{aligned} \phi(0) &= e_0^* \\ \phi(n+1) &= \text{let } x \text{ be } \overline{n} \text{ and } y \text{ be } \phi(n) \text{ in } e_1^* \end{aligned}$$

The meta-function ϕ is defined by primitive recursion.

Crucially, the function ϕ is representable in **F**.

1. Define types *Tm*, *Tp*, and *Ctx* using Church encodings. The constructor *case* has type $\mathit{Tm} \rightarrow (\mathit{Nat} \rightarrow \mathit{Tm}) \rightarrow \mathit{Tm}$, which is properly inductive.
2. Define ϕ of type $\mathit{Nat} \rightarrow \mathit{Tm}$, where *Nat* is the type of Church numerals, as above.

So the syntax of $\mathbf{T}_\omega$ is representable in **F** as an inductive type.

Defining E_ω

It is now straightforward to define E_ω for $\mathbf{T}_\omega$ using only structural induction on *Tm*.

1. Compute *canonical* (η-long, β-normal) and *atomic* (head normal) forms for terms, guided by types.
2. No problem with commuting conversions, *etc.*, because of the meta-function representation.

For example, $E_\omega(\text{case}\, e \,\text{of}\, \phi)$ is defined by

1. Let $\overline{n}$ be $E_\omega(e)$. (Canonize e, unquote to obtain n.)
2. Yield $E_\omega(\phi(n))$. (Call $\phi(n)$, canonize result.)

Relies on *hereditary substitution* to maintain canonical form!
This is definable by lexicographic induction on structure of types and terms.

It's All Just Focusing!

The canonical form required is just the *focused* presentation of **T**.

1. E is essentially a proof of completeness of focusing!
2. *Hereditary substitution* is just the proof of cut elimination for focused proofs.

See Licata, Zeilberger, Harper (LICS 2008 forthcoming) for full details.