On the meaning of decidability issues in dependent types for the problem of output correctness

Giuseppe Primiero

Giuseppe.Primiero@UGent.be

July 2, 2009
European Conference on Computing and Philosophy
Philosophy of Computer Science Track

Centre for Logic and Philosophy of Science
Centrum voor Logica en Wetenschapsfilosofie

GPI - Hertfordshire

The background

- B.C. Smith, "Limits of correctness in computers", (1994): *can computer systems satisfy correctly their designers aim?*
    1. the use of models in the construction of computer systems;
    2. levels of abstractions dealt with by models;
    3. partiality of representations by models;
    4. the role of feedback in judging models;

The Question

- Syntactic correctness: *is it possible to formulate correct structural procedures to satisfy given specifications?*
  1. an appropriate language: dependent types (embedded operational semantics + treatment of information sources);
  2. limits of correctness: decidability;
  3. useful extensions: accessibility, feedback, multiple sources;

Dependent Types: some known facts

- Curry-Howard Isomorphism: propositions-as-types and proofs-as-terms identities;
- Dependent Types: extension to the first-order setting, functional language (MLTT; LF);
- The program-meets-specification variant: dependency as routine-subroutines relation;
  - $+$: description of more complex programs;
  - $+$: more precise typing procedure, less bad-behaved terms;
  - -: increasing of computational information: complicated encoding.

Dependent Types (2)

Definition (Language)

- $A, B, \cdots :=$ *types*: specifications of possible values computable by a program;
- $a, b, \cdots :=$ *terms*: instances of programs;
- $\Gamma, \Delta, \cdots :=$ *contexts*: subroutines;
- $a : A :=$ typed term declaration;
- $[x : A] :=$ variable declaration;
- $b : B[x : A]$: dependent terms are interpreted as programs calling subrotines;
- $b : B[x/a : A]$: substitution is the satisfaction of the call at runtime for the dependent routine.

The Language

$A\ type$ Type declaration $\qquad \dfrac{A\ type}{a : A\ type}$ Value Data formation

$$\frac{A\ type}{x : A[x : A]}\ \text{Value Assumption formation}$$

$$\frac{A\ type \quad B\ type[x : A]}{(x : A)B\ type}\ \text{Function formation}$$

$$\frac{c : (x : A)B \quad a : A}{c(c) : B[x : A]}\ \text{Application}$$

The Language (2)

$$\frac{a : A \quad b : B[x : A]}{(x)b : B[x : A]} \text{ Abstraction}$$

$$\frac{a : A \quad b : B}{(a, b) : A \wedge B} \text{ Conjunction}$$

$$\frac{a : A}{l(a) : A \vee B} \quad \frac{b : B}{r(b) : A \vee B} \text{ Disjunction}$$

$$\frac{x : A \vdash b : B(x)}{(x)b : (\forall x : A)B(x)} \text{ Universal quantification}$$

$$\frac{a : A \quad b : B(a)}{(a, b) : (\exists x : A)B(x)} \text{ Existential quantification}$$

A simple Example

- A typed function to sort lists

```
sort:NatList => Sortedlist
let Sortedlist:=
match Natlist with
[] => []
l (x::Natlist) => let Sortedlist := <l,p>
l := Natlist insert p:Sorted l
```

- the type of functions mapping lists of natural numbers to sorted lists of natural numbers

Subtyping: explicit vs. implicit information (cf. Turner (2007))

- Dependent Types as hidden computational information:

  $(\forall x : A, \exists y : B)S(x, y)$ – for each unvalued term x one gets a pair (x, y) depending on x, containing a proof plus related computational information

- Subtypes as explicit counterpart:

  the pair (f, p), with program f and proof p that f is of type $S(f)$, hence the existential type $(\exists x : [A] \Rightarrow [B])S(x)$;

Language with Subtypes

$$\frac{A \ type \qquad B(x) \ type[x:A]}{\{x:A \mid B\} \ type}$$ Subset Formation

$$\frac{a:A \qquad b:B[x/a]}{a:\{x:A \mid B(x)\}}$$ Subset introduction

$$\frac{a:\{x:A \mid B(x)\} \qquad c(x):C(x)[x:A; y:B]}{c(a):C(a)}$$ Subset elimination

$$\frac{a:A[\Gamma] \qquad a:\{x:A \mid B(x)\}[\Gamma]}{b:B[\Gamma]}$$ Dependent Subsumption

Requirements on Program-meets-Specification

1. well-formedness:
   - each involved value is well-formed
     (with subtyping, computation depends predicatively on type formation, impredicatively by universes or kinding);
2. termination:
   - β-η-conversion rules are needed on components terms
     (termination property for routines);

Type-checking

- The efficiency of the program is based on the evaluation mechanism for the system. General formulation of the proof-checking problem:

Definition (Type-checking Problem)

Given a context Γ, term a and type A, is $\Gamma \vdash a : A$ a derivable expression?

Type-checking (2)

- In a dependent type format, accessibility of all $x' : A'$ in Γ is formally expressed as *Type-reconstruction*

Definition (Type-reconstruction Problem)

Given a term a, there exists a type A and a dependency context Γ such that $\vdash a : A[\Gamma]$ is a derivable expression?

Typability and Type-checking in Simple Types

- Typability and type-checking equivalent to unification, decidable properties:

  ex. let $a : A$ and $b : B$, any typing of $x(yb)(y(fa))$ forces $f : A \rightarrow B$.

- Inhabitation: to answer $\Gamma \vdash ? : A$, apply one of the following tactics:
  - For $A = B \rightarrow C$, ask if $\Gamma, B \vdash ? : C$;
  - For $A = C$ pick $B_1 \rightarrow \cdots \rightarrow B_k \rightarrow C$ from Γ, where $k \geq 0$, then ask if $\Gamma \vdash ? : B_i$, for all i.

Typability and Type-checking in Dependent Types

- Type-inhabitation and Type-reconstruction are undecidable properties: require *explicit* accessibility on contextual data;
- Type-checking not perfomed on the type of variables: soundness presupposes well-formed contexts – examples: *Cayenne*, *DependentML*;
- Typability is decidable with β-reduction on all formulas plus a lemma on the reducibility of contexts or dependency-erasing functions (example: λP);

Modal Correctness

- Correctness on input is characterized by a full treatment of computational information;
  - reconstruction on abstracted information / termination on procedures
  - example: the model of completely presented types in Turner (1993)
- further solution: expressing correctness of an algorithm wrt its subroutines accessibility:
  - can all the subroutines be executed at runtime?
  - at which level of subprocesses does the program fail?

Using modal contexts: labelling formulas

- $a : A[\Box(x' : A')]$ = the program a satisfies specification A by calling subroutine for specification A' evaluated at runtime in any context (and can be used safely by any other routine);
- $a : A[\Diamond(x : A')]$ = the program a satisfies specification A by calling subroutine for specification A' evaluated only in the present context (cannot be used safely by other routines).

Here "safely" means "without risk of incurring in loops".

Using modal contexts: labelling formulas

- $a : A[\Box(x' : A')]$ = the program a satisfies specification A by calling subroutine for specification A' evaluated at runtime in any context (and can be used safely by any other routine);

- $a : A[\Diamond(x : A')]$ = the program a satisfies specification A by calling subroutine for specification A' evaluated only in the present context (cannot be used safely by other routines).

Here "safely" means "without risk of incurring in loops".

Remarks on using modal contexts

- the judgmental interpretation of $\Box/\Diamond J$ is not trivial (non propositional);
- meaning dependent on introduction/elimination rules for modalities;
- modalities from context are preserved to index construction of a staged program;
- growing formal literature (ex: Pfenning 2001); applications to code mobility (Moody 1993) and staged computation (Nanevski et al. 2008).

Levels of failure (1)

Internal information failure: "which step in the program execution (routines, calls for sub-routines) fails?"

Definition (Internal Levels Of Failure)

- IL1 correctness by subcalls recursion (accessibility);
- IL2 correctness by termination procedures (evaluation at runtime).

Levels of failure (2)

External information failure: "which data is missing or fails on dependency, so that the termination process fails?"

Definition (External Levels Of Failure)

- IL3 correctness by data dependency (well-formedness on dependency);
- IL4 correctness by data retrieval (failure-with-world).

Interaction

- prevention of program failure is syntactically based on completeness of data;
- control on modal format triggers the issue of human-machine connection as an higher level of reliability;
- further extension: priority relations on terminations.

Conclusion

- “no [...] social process can take place among program verifiers” (De Millo et al. 1979)
- dependent programming offers ways to implement them.