Exceptions in Dependent Type Theory

Jorge Luis Sacchini*

Carnegie Mellon University, Doha, Qatar
sacchini@qatar.cmu.edu

Exceptions provide a convenient mechanism for signaling errors in a program. Signaling is performed by *raising an exception*, which effectively causes a non-local transfer to a dynamically-installed *handler* that can capture the exception and perform some action, e.g. recover from the error situation. If no handler is found, execution is aborted. Most modern programming languages provide built-in mechanisms for raising and handling exceptions.

In dependently-typed programming languages, such as Agda [5], Coq [6], exceptions can be encoded using sum-types. For example, a division function could be given type $\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} + \mathsf{div\_by\_zero}$. By giving such types a monadic structure, an approach popularized by Haskell, programming with encoded exceptions is rather straightforward. One of the benefits of this approach is that exceptions are encoded in types, which means that a compiler can enforce that all exceptions are handled. Furthermore, in the case of Coq and Agda, logical consistency is preserved, as the language is not changed.

On the other hand, having a primitive notion of exceptions has a number of advantages over encoded exceptions. First, primitive exceptions are more convenient for programming. For example, the expression $1 + \mathsf{div}\,1\,0$ is valid (when executed it would raise an exception), while in the encoding given above, we first have to analyze the value returned by div before proceeding with the addition. This affects performance as well, as we have to pack and unpack the encoded exceptions at every use. Second, exceptions have better support for modularity and code reuse [3]. For example, a higher-order function can be passed as argument a function that may raise exceptions without jeopardizing type safety.

Adding support for first-class exceptions in dependent type theory poses several challenges. First, exceptions usually can have any type; for example 0 and $\mathsf{raise}\,\mathsf{div\_by\_zero}$ can have both type nat. This is undesirable in a dependent type theory as it would lead to logical inconsistencies. To overcome this problem, we need to keep track of exceptions in the type system. We follow the approach given by Lebresne [4]. He proposes a type constructor of the form $A \uplus \psi$, where A is a type and ψ is a set of exceptions. This type is essentially a sum type between regular and exceptional values. Hence, 0 has type nat, while $\mathsf{raise}\,\mathsf{div\_by\_zero}$ has type $\mathsf{nat} \uplus \{\mathsf{div\_by\_zero}\}$.

Second, exception impose a fixed evaluation order—usually, but not always, call-by-value (CBV). In a dependent type theory, where we intent to reason about open terms, we do not want to commit to any evaluation order. To solve this problem, David and Mounier [2] and also Lebresne [4] consider *call-by-name* exceptions. The idea is to have the reduction rules:

$$(\lambda x.t)\,u \to t[u/x] \qquad\qquad (\mathsf{raise}\,\varepsilon)\,u \to \mathsf{raise}\,\varepsilon$$

So that $(\lambda x.t)\,(\mathsf{raise}\,\varepsilon)$ reduces to $t[\mathsf{raise}\,\varepsilon/x]$. This reduction rules, including the usual behavior for handling exceptions, result in a confluent relation.

Third, we mentioned that one advantage of the exception mechanism is better support for modularity and code reuse. The typical example is a sorting function that takes a comparison

*This work was made possible by a NPRP grant (NPRP 09-1107-1-168) from the Qatar National Research Fund (a member of The Qatar Foundation). The statements made herein are solely the responsibility of the author.

function of type, let us say, $\mathsf{nat} \to \mathsf{nat} \to \mathsf{bool}$. If we pass an argument of type $\mathsf{nat} \to \mathsf{nat} \to \mathsf{bool} \uplus \{\varepsilon\}$, we expect that the result will still be well typed, even if it may raise exception ε when applied to a list (i.e. we expect the result to be of type $\mathsf{list\,nat} \uplus \{\varepsilon\}$). Lebresne [4] introduces *corrupted types* to allow this behavior. A corrupted type has the form A^ψ, where ψ is a set of exceptions. A term of type A^ψ can be seen as a term of type A where some subexpression is replaced by $\mathsf{raise}\,\varepsilon$ (with $\varepsilon \in \psi$). Corruption has nice properties like distribution across function types: $(A \to B)^\psi = A^\psi \to B^\psi$; this means that a sorting function can have type $(\mathsf{nat}^\psi \to \mathsf{nat}^\psi \to \mathsf{bool}^\psi) \to \mathsf{list}^\psi\,\mathsf{nat} \to \mathsf{list}^\psi\,\mathsf{nat}$, for any ψ (including $\emptyset$), effectively allowing reuse of the function in the presence of exceptions.

In general, we expect that, in any well-typed program of A, replacing any subexpression by $\mathsf{raise}\,\varepsilon$ would result in a well-typed program of type A^ψ. However, this works in simply-typed systems, but not in the presence of dependent types. For example, consider a function P of dependent type

$$\begin{array}{l}\Pi x{:}\mathsf{nat}.\ \mathsf{case}\,(\mathsf{try}\ x\ \mathsf{ow}\ \varepsilon \Rightarrow \mathsf{S}\,\mathsf{O})\ \mathsf{of}\\ \quad \mid \mathsf{O} \Rightarrow \mathsf{nat} \to \mathsf{nat}\\ \quad \mid \mathsf{S}\,x \Rightarrow \mathsf{nat}\end{array}$$

Then $P\,\mathsf{O}\,\mathsf{O}$ has type nat, but $P\,(\mathsf{raise}\,\varepsilon)\,\mathsf{O}$ is not well typed. Although this example is a bit artificial, it shows that corruption cannot be applied freely in the presence of dependent types.

In this proposed talk, we will present $\lambda^{\Pi,\varepsilon}$, a predicative type theory with inductive types and call-by-name exceptions. The type system features union types of the form $T \uplus \psi$ to account for exceptions at top level, and corrupted types of the form I^ψ, for inductive types I, meaning that an exception can occur under constructors. $\lambda^{\Pi,\varepsilon}$ enjoys desirable metatheoretical properties such as subject reduction and strong normalization (proved using a modified Λ-set model [1]).

However, for the reasons explained above, $\lambda^{\Pi,\varepsilon}$ does not feature a full corruption operator. This implies that, although $\lambda^{\Pi,\varepsilon}$ is more convenient to use than encoded exceptions, it still does not have all the advantages of using primitive exceptions. In this talk, we will also discuss the limitations of $\lambda^{\Pi,\varepsilon}$ and possible ways to overcome them.

References

[1] Thorsten Altenkirch. *Constructions, Inductive Types and Strong Normalization.* PhD thesis, University of Edinburgh, November 1993.

[2] R. David and G. Mounier. An intuitionistic Λ-calculus with exceptions. *J. Funct. Program.*, 15(1):33–52, January 2005.

[3] Simon L. Peyton Jones, Alastair Reid, Fergus Henderson, C. A. R. Hoare, and Simon Marlow. A semantics for imprecise exceptions. In Barbara G. Ryder and Benjamin G. Zorn, editors, *PLDI*, pages 25–36. ACM, 1999.

[4] Sylvain Lebresne. A system F with call-by-name exceptions. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP (2)*, volume 5126 of *Lecture Notes in Computer Science*, pages 323–335. Springer, 2008.

[5] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers University of Technology, 2007.

[6] The Coq Development Team. *The Coq Reference Manual, version 8.3*, 2009. Distributed electronically at `http://coq.inria.fr/doc`.