A Machine-assisted Proof of the Subject Reduction Property for a Small Typed Functional Language[1]

Ana Bove Universidad de la República, Montevideo, Uruguay[2]
Alvaro Tasistro Chalmers University, Gothenburg, Sweden[3]

We are interested in the application of constructive type theory to the theory of programming languages. By constructive type theory we understand primarily the formulation of Martin-Löf's set theory using the theory of types as logical framework ([ML 87, Nor 90]). This has been conceived as a formal language in which to carry out constructive mathematics. So what we want to do in the first place is to investigate constructive formalizations of the mathematics of programs.

Constructive type theory can also be viewed as a programming language. In type theory we write types and objects of these types, and the objects can be seen as functional programs. In particular, propositions are interpreted as types whose objects are the proofs of the propositions in question. Then, a proof of a theorem becomes in general a function that accepts as arguments individuals or proofs corresponding to the assumptions of the theorem and computes a proof of the conclusion. When the theorem states the existence of an object with certain properties, its proof will compute such an object from the corresponding assumptions. In this way, many important algorithms used in the implementation of programming languages arise naturally as proofs of properties of the languages. In other words, the formalization of the relevant parts of the theory of programming languages gives implementations of these languages that are correct by construction. A main motivation of our work is to actually carry out this idea in practice, that is, to investigate the production of verified implementations of programming languages.

Writing programs in constructive type theory amounts to writing completely formal proofs of often complex theorems. This makes the practical applicability of type theory depends strongly on the availability of adequate programming environments (alias proof assistants). A number of these systems have been or are being developed. We expect our work to become a useful experience in this connection too.

In the present work, we conduct a first experiment along the lines given above. We consider a small polymorphic functional language and write a formal proof in constructive type theory of the main property relating the type system of the language with the evaluation of its expressions, namely the *Subject Reduction Property*. We use the proof assistant ALF ([Alt 94]).

The abstract syntax of the language considered is the following :

$$e ::= x \mid \lambda x\,.\,e \mid (d\ \ e) \mid \mathbf{fix}\ x\,.\,e \mid true \mid false \mid \mathbf{if}\ d\ \mathbf{then}\ e\ \mathbf{else}\ f$$

We consider an (operational) evaluation semantics, i.e. an inductive definition of what it means for an expression to have another expression as value. As is often the case in actual implementations of functional languages, only closed expressions are intended to be evaluated, and no evaluation takes place within the scope of a binding operator. As a consequence, variable capture cannot occur during evaluation which allows a very simple definition of substitution.

The values of the language are expressions of the form $true$, $false$ or $\lambda x\,.e$. The semantics determines a partial function from expressions to values. Among the closed expressions without a value, there are for instance $\mathbf{if}\ \lambda x\,.\,x\ \ \mathbf{then}\ true\ \ \mathbf{else}\ false$ and $\mathbf{fix}\ x\,.\,x$. The first one is intuitively a type incorrect expression, whereas the second one is "eternally looping".

The types are given by the following abstract syntax : $\alpha ::= \mathbf{Bool} \mid \alpha \rightarrow \beta$

The system of assignment of types to expressions is adapted from the definition of Pure Type Systems in [Geu 90]. It proves judgements of the form $\Gamma \vdash e : \alpha$ where e is an expression, α a type and Γ a so-called context. Contexts are lists of *declarations* of the form $x : \alpha$ in which each variable is declared at most once. In correct judgements $\Gamma \vdash e : \alpha$, the free variables of e are declared in Γ .

[1] Partially supported by a Bid-Conicyt grant and by a Swedish Institute scholarship
[2] bove@fing.edu.uy, http://www.fing.edu.uy/~bove
[3] tato@cs.chalmers.se

The system contains one rule for each form of expression plus a thinning rule. An important point is that we make no "variable conventions" allowing to consider representatives of classes of α-equivalent expressions in which bound variables are conveniently chosen, as is done for instance, in [Bar 92]. So we have to deal explicitly with the possibility that within the scope of a bound variable the same variable is used in another abstraction. Expressions in which such a situation occurs may still receive types in our system, for which purpose the thinning rule is essential.

The main result relating well-typedness to evaluation is the *Subject Reduction* property : *If a closed expression* e *of type* α *has a value, this value is also a closed expression of type* α .

In our case, this property only applies to closed expressions. This is because evaluation of open expressions may lead to variable capture.

For proving subject reduction we need the *Substitution Lemma* : *Let* $[\Gamma\ ,\ x\ :\alpha\ ]$ *and* Δ *be two contexts and* $\_\ \&\ \_$ *the function that concatenates two contexts. Let* d *and* e *be expressions,* x *a variable and* α *and* β *types. If we can derive* $[\Gamma\ ,\ x\ :\alpha\ ]\ \&\ \Delta\ \vdash e\ :\beta$ *and* $\Gamma\ \vdash d\ :\alpha$, *then we can also derive* $\Gamma\ \&\ \Delta\ \vdash e\,[d/x]\ :\beta$.

The whole development has been written in a completely formal manner and type checked using the proof assistant ALF. The complete code is presented in [Bov 95]. The formalization in ALF proceeded quite smoothly. Interesting discussions arise in connection to alternative formalizations at various points in the development, most notably the definition of the notion of context and the formulation of the substitution lemma. One point that turned out to be unexpectedly complicated was the formulation of the type system. Standard references in the literature often treat contexts as finite sets (in the sense of classical set theory) and make conventions as to the use of free and bound variables, all of which needs to receive an appropriate reformulation when we require complete formalization in constructive type theory. Actually it will be the matter of further work to investigate alternative formulations of the type system.

A natural continuation of this work is to consider the existence of principal type schemes, whose proof in type theory gives an algorithm of type inference. This will then give a first verified piece of an implementation of the language considered.

Machine assisted proofs of properties of typed lambda calculi are presented in [Pol 94] and [Coq 94]. There are several interesting differences between our approach and those of these works, which we cannot comment on here due to space limitations. In [Pfe 91], a formalization of the operational semantics and type system of a functional language in the logic programming language *Elf* is given. This formalization provides an interpreter for the language but, on the other hand, proofs of properties as we have considered cannot be fully represented.

References

[Alt 94] *A User's guide to ALF.* T. Altenkirch, V. Gaspes, B. Nordström, B. von Sydow. Department of Computer Science, University of Göteborg / Chalmers, Sweden. June, 1994.

[Bar 92] *Lambda Calculi with Types.* H. P. Barendregt. In Handbook of Logic in Computer Science, Vol. II. Gabbai, Abramsky and Maibaum editors. Oxford University Press. 1992.

[Bov 95] *A Machine-assisted Proof of the Subject Reduction Property for a Small Typed Functional Languege.* Master thesis. Technical Report of the Department of Computer Science, University of the Republic, Uruguay. To appear, 1996.

[Coq 94] *Type Theory and Programming.* T. Coquand, B. Nordström, J. M. Smith, B. von Sydow. EATCS Bulletin. February, 1994.

[Geu 90] *Type Systems for Higher Order Logic.* J. H. Geuvers. Technical Report of the Department of Computer Science, Catholic University, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands. 1990.

[ML 87] *Philosophical Implications of Type Theory.* Lectures given at the Facoltá de Lettere e Filosofia, Universitá degli Studi di Firenze, Florence, March 15th. - May 15th., 1987. Privately circulated notes.

[Nor 90] *Programming in Martin - Löf's Type Theory. An Introduction.* B. Nordström, K. Petersson, J. M. Smith. Oxford University Press. 1990.

[Pfe 91] *Logic Programming in the LF Logical Framework.* F. Pfenning. In *Logical Frameworks.* Cambridge University Press. G. Huet and G. Plotkin Eds. 1991.

[Pol 94] *The Theory of LEGO : A Poof Checker for the Extended Calculus of Constructions.* R. Pollack. Doctor of Philosophy Thesis. University of Edinburgh. Available by anonymous ftp from `ftp.cs.chalmers.se` in directory `pub/users/pollack`. 1994.