A critical issue concerning programming languages and proof assistants based on dependent type theory is the efficient implementation of reduction on open terms, where meta-variables can appear: this question is important not only for the execution of dependently typed programs, but already in the typechecker implementing the validation of such programs, through the use of the conversion rule. Following the Curry-Howard tradition, we are currently investigating the relation between the internal language of Agda and a focused sequent calculus presentation of intuitionistic logic, the sequent calculus being naturally well-suited for typing open terms.

1 Internal Languages using Dependent Types

Different approaches can be used to implement a dependently typed language. For example, the proof terms used in the Coq proof assistant are really based on the natural deduction presentation of a higher-order intuitionistic logic, where matching is performed through a dedicated construct. However, in the Agda language, matching is part of an equational style of programming where the left-hand side of a definition can have a complex shape. As a result, the internal language used in Agda is not exactly a dependently typed pure \(\lambda\)-calculus, but a calculus containing the case-splitting tree of a definition involving matching.

A close look at the internal language of Agda reveals that it is much related to a presentation of intuitionistic logic in the sequent calculus. In particular, one can see the use of a definition as applying the cut rule, where one premise corresponds to the definition and the other to its use inside another term. Then, whenever the typechecker needs to compare two terms, it needs to reduce them by unfolding the definitions, which corresponds to a cut elimination process — note that all cuts cannot be eliminated, because of recursive definitions. At this point, it is important to observe that cut elimination in the sequent calculus, although equivalent to \(\beta\)-reduction in the standard \(\lambda\)-calculus, can be performed in many different ways. Moreover, important insights on cut elimination have been obtained through the analysis of linear logic and its intuitionistic variant. Based on the idea that more control can be gained over normalisation of terms in the sequent calculus, compared to natural deduction and the associated \(\beta\) rule, we propose an analysis of reduction in Agda using the tools of the sequent calculus to improve the efficiency of reduction, and therefore also the efficiency of typechecking.

2 Term Assignments for Focused Sequent Calculi

Although it offers a fine-grained representation of proofs, the sequent calculus LJ for intuitionistic logic is not well-suited for computation, in its basic, unrestricted form. Indeed, it lacks a canonical structure as found in natural deduction, that can guide the reduction process, but a well-behaved form can be recovered through the focusing technique. This strong structure is also important to compare proof terms, which is essential in a proof assistant — in the setting of

*Supported by grant 10-092309 from the Danish Council for Strategic Research to the Demtech project.
natural deduction, $\beta\eta$-normal forms are considered, while in the sequent calculus one should consider cut-free focused proofs. Moreover, focusing allows to consider only the fragment of proofs isomorphic to the standard $\lambda$-calculus, or to allow for proofs containing more sharing.

The internal language of Agda uses an application to list of arguments called spines, and is best compared to the $\lambda$-calculus of Herbelin [5], the computational interpretation of LJT, a particular focused system related to call-by-name reduction in the $\lambda$-calculus. There are other possible choices in the design of a focused system, for example following a call-by-value approach, but LJT is an excellent starting point — in particular, because it is the basis for a system developed to handle dependent types in the sequent calculus [6]. Our claim is that the versatile structure of the sequent calculus, associated with the strong structure provided by focusing, makes it an ideal framework to study, from a proof-theoretical perspective, the implementation of dependently typed languages.

As an example, consider a focused system where $\vdash$ denotes the inversion phase and $\models$ denotes the focusing phase. One can handle case-splitting on natural numbers using the following rules:

$$
\Gamma \vdash t : C \quad \Gamma, \Gamma' \{\text{zero} / x \} \vdash T : C\{\text{zero} / x \} \\
\Gamma, n : \mathbb{N}, \Gamma' \{\text{suc} n / x \} \vdash T' : C\{\text{suc} n / x \}
$$

and this yields the following interpretation of an equational definition of addition, as it would be written in Agda:

$$
\text{add zero } n = n \\
\text{add (suc } m') n = \text{suc (add } m' \cdot n)
\Rightarrow m : \mathbb{N}, n : \mathbb{N} \models \text{split } (m, \mapsto \cdot n, m' \mapsto \text{suc (add (} m' \cdot : n \cdot : \varepsilon) \cdot : \varepsilon))
$$

From the basis of a sequent system adding dependent types to $\lambda$, we can start the investigation of more complex focused systems, as obtained for example when considering the full LJF calculus of [7], of which the usual LJT and LQJ calculi are fragment. The goal of this move from the standard setting of natural deduction to the sequent calculus is to exploit the ability of the sequent calculus to capture different evaluation strategies, such as call-by-name and call-by-value, in a single, unifying framework, and therefore allowing to improve the efficiency of reduction.

References


