EFFICIENT EVALUATION WITH CONTROLLED DEFINITION UNFOLDING

ANDRÁS KOVÁCS

In polymorphic and dependently typed languages, there is a tension between checking conversion as efficiently as possible, and preserving unfoldings of definitions. The latter is desirable in user interaction, and also critically important in unification, where metavariable solutions tend to blow up in size if definitions are unfolded all the time. Additionally, conversion checking and scope checking can benefit from *speculative shortcuts*, where we attempt to check something without unfolding certain definitions, and possibly retry with unfolding, on failure.

We describe a design for efficient open evaluation with preserved unfoldings. The simplest setup distinguishes top-level and local definitions, and only preserves top-level unfoldings; we describe this version, although preserving arbitrary unfoldings is also possible with a moderate increase in complexity. We build on a standard normalization-by-evaluation setup, where terms are evaluated in a semantic domain during elaboration. Below is a Haskell sketch for pure lambda calculus with top-level definitions. We work in `Strict` Haskell, so only ∼-marked data fields are lazy.

```
data Tm    = TopDef Name | LocalVar Name | App Tm Tm | Lam Name Tm
data Val   = VUnfold Name Spine ~Val | VNe Name Spine | VLam Name (Val → Val)
data Spine = Empty | SpineApp Spine Val
```

The whole program is a list of top-level defintions, and inside terms the TopDef construction refers to those. The main novelty is VUnfold in values. VUnfold represents a delayed unfolding choice. When we process it or convert it back to a term, we can choose "no unfolding" by only examining the stored name and the spine, or "unfolding" by forcing the ∼Val. We sketch evaluation.

```
vapp :: Val → Val → Val
vapp (VUnfold x sp v) u = VUnfold x (SpineApp sp u) (vapp v u)
vapp (VNe x sp) u       = VNe x (SpineApp sp u)
vapp (VLam _ f) u       = f u

eval :: [(Name, Val)] → [(Name, Val)] → Tm → Val
eval topvs vs (TopDef x)   = VUnfold x Empty (lookup x topvs)
eval topvs vs (LocalVar x) = lookup x vs
eval topvs vs (App t u)    = vapp (eval topvs vs t) (eval topvs vs u)
eval topvs vs (Lam x t)    = VLam x (λ v → eval topvs ((x, v):vs) t)
```

The evaluation of top-level names has a significant overhead, but it's usually only a constant factor, and we get a large amount of shared computation and structure. In particular, we almost never need to re-evaluate terms with different unfolding options.

In practical implementations, we need a bit more machinery:

- A *forcing* function reduces a value to head-normal form by peeling off all unfoldings, until we hit a value that's not an unfolding. This is required in elaboration and in unification.
- To exactly preserve unfoldings, in some places we need to compute with *pairs of values* where the two values are convertible, but one value is "minimally" unfolded and the other one is "maximally" unfolded.

Implementation. `smalltt` [Kov23a], `sett` [KB23], cctt [Kov23b] and `sixty` [Fre23] use variations of the above setup. The basic idea was first used in early versions of `smalltt`, then revised by Fredriksson in `sixty`, then further developed in `smalltt`, `sett` and `cctt`.

References

[Fre23] Olle Fredriksson, 2023. URL: https://github.com/ollef/sixty.

[KB23] András Kovács and Rafaël Bocquet, 2023. URL: https://github.com/AndrasKovacs/sett.

[Kov23a] András Kovács, 2023. URL: https://github.com/AndrasKovacs/smalltt.

[Kov23b] András Kovács, 2023. URL: https://github.com/AndrasKovacs/cctt.