Deterministic Parallelism

Vikraman Choudhury

April, 2018

1 Deterministic Parallelism

LVish as described in Kuper et al. [2014] is a deterministic parallel programming library which provides the advantages of parallel speedup while avoiding the nondeterministic bugs that plague concurrent code. This is made possible by the use of *LVars* which provides a shared memory construct with well-defined semantics. However, the implementation of LVish relies on dynamic enforcement of such guarantees, which makes it difficult to formalize the static and dynamic semantics of the language.

We formalize a toy language with a single *IVar* (instead of LVars) which is deterministic by construction. The language is deeply embedded in Agda, and admits an interpretation to the host language. The formalization follows the style used in the Dependently Typed Metaprogramming notes by McBride [2013].

1.1 Syntax

The language is a simply typed lambda calculus, with a base type, products and exponentials.

```
data ★ : Set where
  ι : ★
  _⊗_ : ★ → ★ → ★
  _⇒_ : ★ → ★ → ★
```

Contexts are defined as snoc-lists.

```
data Cx (A : Set) : Set where
  ε : Cx A
  _,_ : Cx A → A → Cx A
```

We setup a De Brujin index over contexts to track bound identifiers.

```
data _∈_ (τ : ★) : Cx ★ → Set where
  here : ∀ {Γ} → τ ∈ Γ , τ
  there : ∀ {Γ σ} → τ ∈ Γ → τ ∈ Γ , σ
```

A store, parameterized over a type, is either empty or full, containing something of that type.

```
data St (A : Set) : Set where
  ⟨−⟩ : St A
  ⟨_⟩ : (a : A) → St A
```

Next, we setup a typing judgment of the following shape $\Sigma \lhd \Gamma \vdash \tau \rhd \Xi$, where Σ and Ξ are stores, Γ is the context, and τ the type. The intended meaning is that, the derivation of τ in the context Γ changes the store state from Σ to Ξ. The typing relation is defined as a logical relation indexed on the type alongwith the pre and post stores. Due to a technical limitation of the LaTeX rendering mechanism in Agda, we have to define the term Tm but it is an alias for the same operator.

```
data Tm (Γ : Cx ★) : St ★ → ★ → St ★ → Set where
```

The typing rules are defined in a syntax-directed manner. If a type is bound in the context, the derivation is immediate, with no change to the store.

```
  var : ∀ {τ Σ}
      → τ ∈ Γ
      ----------------
      → Σ ◁ Γ ⊢ τ ▷ Σ
```

A bound type in the extended context is abstracted by a function type, and a function type can be applied, with no change to the store.

```
  lam : ∀ {τ σ Σ}
      → Σ ◁ Γ , σ ⊢ τ ▷ Σ
      ----------------
      → Σ ◁ Γ ⊢ σ ⇒ τ ▷ Σ

  app : ∀ {τ σ Σ}
      → Σ ◁ Γ ⊢ σ ⇒ τ ▷ Σ
```

```
      → Σ ◁ Γ ⊢ σ ▷ Σ
      ----------------
      → Σ ◁ Γ ⊢ τ ▷ Σ
```

Products are introduced and eliminated as usual, and do not change the store.

```
prd : ∀ {τ σ Σ}
      → Σ ◁ Γ ⊢ σ ▷ Σ
      → Σ ◁ Γ ⊢ τ ▷ Σ
      ----------------
      → Σ ◁ Γ ⊢ σ ⊗ τ ▷ Σ

fst : ∀ {τ σ Σ}
      → Σ ◁ Γ ⊢ σ ⊗ τ ▷ Σ
      ----------------
      → Σ ◁ Γ ⊢ σ ▷ Σ

snd : ∀ {τ σ Σ}
      → Σ ◁ Γ ⊢ σ ⊗ τ ▷ Σ
      ----------------
      → Σ ◁ Γ ⊢ τ ▷ Σ
```

A get operation can only be done on a full store. This is enforced by a full store for the start state. The second argument can access the read value in the extended context.

```
get : ∀ {τ σ Σ}
      → ⟨ σ ⟩ ◁ Γ , σ ⊢ τ ▷ Σ
      ----------------
      → ⟨ σ ⟩ ◁ Γ ⊢ τ ▷ Σ
```

A put operation can only be done on an empty store. This is enforced by an empty store for the start state. The first argument of put updates the state, and the updated state is available to the second argument.

```
put : ∀ {τ σ Σ}
      → ⟨ - ⟩ ◁ Γ ⊢ σ ▷ ⟨ - ⟩
      → ⟨ σ ⟩ ◁ Γ ⊢ τ ▷ Σ
      ----------------
      → ⟨ - ⟩ ◁ Γ ⊢ τ ▷ Σ
```

For a fork operation, two threads can run concurrently only if they make compatible updates to the store. There are two symmetric cases, one thread does not update the store, while the other thread does, so these threads can be successfully joined. This is expressed by a single rule, and the product of the result of the two computations is returned.

```
fork : ∀ {τ σ Σ Ξ}
     → Σ ◁ Γ ⊢ σ ▷ ⟨-⟩
     → Σ ◁ Γ ⊢ τ ▷ Ξ
     ----------------
     → Σ ◁ Γ ⊢ σ ⊗ τ ▷ Ξ
```

Although not strictly necessary for the interpreter, we can implement substitution for this calculus for sanity-checking. First we define the friendly fish operator for context extension with a list.

```
_<><_ : ∀ {A} → Cx A → List A → Cx A
Γ <>< [] = Γ
Γ <>< (a :: as) = (Γ , a) <>< as
```

Substitutions are morphisms of contexts.

```
Sub : Cx ★ → Cx ★ → Set
Sub Γ Δ = ∀ {τ Σ Ξ} → τ ∈ Γ → Σ ◁ Δ ⊢ τ ▷ Ξ

Shub : Cx ★ → Cx ★ → Set
Shub Γ Δ = ∀ Π → Sub (Γ <>< Π) (Δ <>< Π)
```

Substitution on terms is type-preserving and acts functorially.

```
subst : ∀ {Γ Δ τ Σ Ξ} → Shub Γ Δ
      → Σ ◁ Γ ⊢ τ ▷ Ξ → Σ ◁ Δ ⊢ τ ▷ Ξ
subst θ (var i) = θ [] i
subst θ (lam t) = lam (subst (λ ρ → θ (_ :: ρ)) t)
subst θ (app f s) = app (subst θ f) (subst θ s)
subst θ (prd s t) = prd (subst θ s) (subst θ t)
subst θ (fst t) = fst (subst θ t)
subst θ (snd t) = snd (subst θ t)
subst θ (get f) = get (subst (λ ρ → θ (_ :: ρ)) f)
```

```
subst θ (put p t) = put (subst θ p) (subst θ t)
subst θ (fork s t) = fork (subst θ s) (subst θ t)
```

We consider a few examples of programs in this calculus that successfully type-check. We begin by fixing types τ and σ.

```
module Examples (τ σ : ★) where
```

e_1 reads a value from the store and applies it to the function in the context.

```
e₁ : ⟨ σ ⟩ ◁ ε , σ ⇒ τ ⊢ τ ▷ ⟨ σ ⟩
e₁ = get (app (var (there here)) (var here))
```

e_2 computes a value by function application and writes it to the store, and the return value is read back.

```
e₂ : ⟨−⟩ ◁ (ε , σ ⇒ τ) , σ ⊢ τ ▷ ⟨ τ ⟩
e₂ = put (app (var (there here)) (var here))
         (get (var here))
```

e_3 runs two threads which compute different values while only one of them updates the store.

```
e₃ : ⟨−⟩ ◁ (ε , σ ⇒ τ) , σ ⊢ τ ⊗ σ ▷ ⟨ τ ⟩
e₃ = fork (app (var (there here)) (var here))
          (put (app (var (there here)) (var here))
               (var here))
```

1.2 Semantics

Now, we can define a semantics for each syntactic construct, by interpreting them back to the host language. The base type is interpreted to the type of natural numbers, and the product and exponential are interpreted to the corresponding types in Agda.

```
⟦_⟧★ : ★ → Set
⟦ ι ⟧★ = ℕ
⟦ σ ⊗ τ ⟧★ = ⟦ σ ⟧★ × ⟦ τ ⟧★
⟦ σ ⇒ τ ⟧★ = ⟦ σ ⟧★ → ⟦ τ ⟧★
```

A context of types can be interpreted back, by abstracting over the interpretation of types.

```
⟦_⟧Cx : {A : Set} → Cx A → (A → Set) → Set
⟦ ε ⟧Cx v = ⊤
⟦ Γ , σ ⟧Cx v = ⟦ Γ ⟧Cx v × v σ
```

De Brujin indices are interpreted by doing a recursive lookup in the context.

```
⟦_⟧∈ : ∀ {Γ τ v} → τ ∈ Γ → ⟦ Γ ⟧Cx v → v τ
⟦ here ⟧∈ (γ , t) = t
⟦ there i ⟧∈ (γ , s) = ⟦ i ⟧∈ γ
```

Stores are interpreted similarly using the interpretation of the type.

```
⟦_⟧St : {A : Set} → St A → (A → Set) → Set
⟦ ⟨−⟩ ⟧St v = ⊤
⟦ ⟨ σ ⟩ ⟧St v = v σ
```

Finally, we give semantics to the typing judgment. We interpret the states using a state monad, where the start state is Σ and the end state is Ξ.

```
⟦_⟧⊢ : ∀ {Γ τ Σ Ξ}
       → Σ ◁ Γ ⊢ τ ▷ Ξ
       → ⟦ Γ ⟧Cx ⟦_⟧★
       → ⟦ Σ ⟧St ⟦_⟧★ → ⟦ Ξ ⟧St ⟦_⟧★ × ⟦ τ ⟧★
⟦ var i ⟧⊢ γ st = st , ⟦ i ⟧∈ γ
⟦ lam t ⟧⊢ γ st = st , λ s → proj₂ (⟦ t ⟧⊢ (γ , s) st)
⟦ app f s ⟧⊢ γ st = st , proj₂ (⟦ f ⟧⊢ γ st) (proj₂ (⟦ s ⟧⊢ γ st))
⟦ prd s t ⟧⊢ γ st = st , (proj₂ (⟦ s ⟧⊢ γ st) , proj₂ (⟦ t ⟧⊢ γ st))
⟦ fst t ⟧⊢ γ st = st , proj₁ (proj₂ (⟦ t ⟧⊢ γ st))
⟦ snd t ⟧⊢ γ st = st , proj₂ (proj₂ (⟦ t ⟧⊢ γ st))
⟦ get f ⟧⊢ γ st = ⟦ f ⟧⊢ (γ , st) st
⟦ put p t ⟧⊢ γ st = ⟦ t ⟧⊢ γ (proj₂ (⟦ p ⟧⊢ γ tt))
⟦ fork s t ⟧⊢ γ st = let (st′ , vt) = ⟦ t ⟧⊢ γ st
                         (_ , vs) = ⟦ s ⟧⊢ γ st
                     in st′ , (vs , vt)
```

Since we have a provably terminating, total function which interprets each term in the calculus, we know that the evaluation is deterministic! We can try to run our examples with the interpreter and get expected results.

```
open Examples ι ι

e₁↦ : ⟦ e₁ ⟧⊢ (tt , (λ n → 3 * n)) 19 ≡ (19 , 57)
e₁↦ = refl

e₂↦ : ⟦ e₂ ⟧⊢ ((tt , (λ n → 3 * n)) , 19) tt ≡ (57 , 57)
e₂↦ = refl

e₃↦ : ⟦ e₃ ⟧⊢ ((tt , (λ n → 3 * n)) , 19) tt ≡ (57 , 57 , 19)
e₃↦ = refl
```

1.3 Conclusion

Using the decidable typechecking algorithm in Agda, we're able to typecheck programs in this language, which implies that we can extract a typechecking algorithm for a real implementation of this language. Alternatively, this could be implemented by deep embedding into Haskell using its recent dependently typed programming features as descibed in Weirich et al. [2017], or using refinement types in Liquid Haskell from Vazou et al. [2017]. It should also be possible to extend this technique to implement the full LVars calculus.

References

Lindsey Kuper, Aaron Turon, Neelakantan R Krishnaswami, and Ryan R Newton. Freeze after writing: Quasi-deterministic parallel programming with lvars. *ACM SIGPLAN Notices*, 49(1):257–270, 2014.

Conor McBride. Dependently typed metaprogramming (in agda). 2013.

Stephanie Weirich, Antoine Voizard, Pedro Henrique Avezedo de Amorim, and Richard A Eisenberg. A specification for dependent types in haskell. *Proceedings of the ACM on Programming Languages*, 1(ICFP):31, 2017.

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement reflection: Complete verification with smt. *Proc. ACM Program. Lang.*, 2(POPL):53:1–53:31, December 2017. ISSN 2475-1421. doi: 10.1145/3158141. URL http://doi.acm.org/10.1145/3158141.