

Singleton Types Here, Singleton Types There, Singleton Types Everywhere

Stefan Monnier & David Haguenaer

Université de Montréal

Dependent types for programmers

Type based formal methods for security

Type based formal methods for reliability

Proof systems for program verification

Proof systems for meta theory

Love & hate

Dependent types and programming languages's troubled relationship:

Dependent types are sexy and elegant

They are to plain types, what plain types are to dynamic typing

They mix very poorly with real-world features

⇒ Poor man's dependent types

Compiler writers do not know what to do with them

⇒ Don't compile

⇒ Drop types

⇒ Drop dependencies

Contributions

Tame dependent types for compilers

Type-preserving closure conversion of CC

Singleton types as powerful as dependent types

Way to make singleton types prettier

Singleton types

Poor man's dependent types which enjoy a phase distinction

$$\text{aref} : n < m \Rightarrow \text{Snat } n \rightarrow \text{Array } \alpha \ m \rightarrow \alpha$$

What about operations on integers?

$$\text{Nat} = \exists n. \text{Snat } n$$

$$+ : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$$

or

$$+ : \text{Snat } n \rightarrow \text{Snat } m \rightarrow \text{Snat } (n + m)$$

¡Two additions!

Poisonous duplication

Everything tends to get sucked up in the copy machine

- First basic data types
- Then operations on them

Proofs at type level want extensions to the type language

- The $n < m$ cannot always be proved automatically
- If the type language does not help, use ... singleton types!

Dependent types

$$\mathit{aref} : \prod t : *, n : \mathit{Nat}, m : \mathit{Nat}. (n < m) \rightarrow \mathit{Array} \ t \ m \rightarrow t$$

No funny Snat , no duplication: simpler, cleaner

Types, values, proofs are neatly intertwined

A happy family ... or a big mess

No more *phase distinction*

Compiling dependent types

Drop types altogether

Drop dependencies

$aref : \forall \alpha : \Omega. \text{Nat} \rightarrow \text{Nat} \rightarrow \text{LT} \rightarrow \text{Array } \alpha \rightarrow \alpha$

⇒ Back to plain types

⇒ “Type preserving” but without preserving type info

There's hope

Conversion to singleton types

Automate the duplication, so the programmer does not have to see it

Split each element into a value and a type

$$n : \text{Nat} \quad \Rightarrow \quad \check{n} : \text{Snat } \hat{n}$$

We can choose $\hat{n} \equiv n$

For other types:

$$e : \tau \quad \Rightarrow \quad \mathcal{C}[[e]] : \mathcal{S}[[\tau]] e$$

Destination type language \supseteq source language

We compile the Calculus of Constructions to λ_H

[Shao, Saha, Trifonov, Papaspyrou, *Type safe certified binaries*, POPL'02]

A simple functional language, where CIC is used for the type language:

$$(exp) e ::= x \mid n \mid e e \mid e[A] \mid \lambda x:A.e \mid \Lambda X:A.f$$

The type of types is an inductive definition:

$$\text{Inductive } \Omega : * := \text{Snat} : \text{Nat} \rightarrow \Omega$$

$$\mid \rightarrow \! \! \rightarrow : \Omega \rightarrow \Omega \rightarrow \Omega$$

$$\mid \forall : \Pi k:*. (k \rightarrow \Omega) \rightarrow \Omega$$

CIC types are λ_H kinds; CIC kinds are λ_H kind schemas

Generalize to functions

Applying previous formula $\mathcal{C}[[e]] : \mathcal{S}[[\tau]] e$:

$$\begin{aligned} (+) : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} &\Rightarrow \mathcal{C}[[(+)] : \mathcal{S}[[\text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}]] (+) \\ \text{sort} : \text{List} \rightarrow \text{List} &\Rightarrow \mathcal{C}[[\text{sort}]] : \mathcal{S}[[\text{List} \rightarrow \text{List}]] \text{sort} \end{aligned}$$

And since we want:

$$\mathcal{C}[[(+)] : \forall n, m : \text{Nat}. \text{Snat } n \rightarrow \text{Snat } m \rightarrow \text{Snat } (n + m)$$

$$\mathcal{C}[[\text{sort}]] : \forall l : \text{List}. \text{Slist } l \rightarrow \text{Slist } (\text{sort } l)$$

We get:

$$\mathcal{S}[[\tau_1 \rightarrow \tau_2]] f \equiv \forall x : \tau_1. \mathcal{S}[[\tau_1]] x \rightarrow \mathcal{S}[[\tau_2]] (f x)$$

$$\mathcal{C}[[\lambda x : \tau. e]] \equiv \Lambda x : \tau. \lambda x' : (\mathcal{S}[[\tau]] x). \mathcal{C}[[e]]$$

What about variables

How to compute $\mathcal{C}[[x]]$?

intensional type analysis or dictionary passing...

What about variables

How to compute $\mathcal{C}[[x]]$?

intensional type analysis or dictionary passing...

Refine $\mathcal{C}[[\lambda x:\tau.e]] \equiv \Lambda x:\tau.\lambda x':(\mathcal{S}[[\tau]] x).\mathcal{C}[[e]]$ into

$$\mathcal{C} c [[\lambda x:\tau.e]] \equiv \Lambda x:\tau.\lambda x':(\mathcal{S}[[\tau]] x).\mathcal{C} \{c, x \mapsto x'\} [[e]]$$

$$\mathcal{C} c [[x]] \equiv c(x)$$

What about type variables

How to compute $\mathcal{S}[[t]]$?

$$head : \prod t : *. List\ t \rightarrow t$$

$$\mathcal{C}[[head]] : \forall t : *. \forall l : List\ t. Slist\ t\ l \rightarrow \mathcal{S}[[t]]\ (head\ t\ l)$$

Becomes

$$\mathcal{C}[[head]] : \forall t : *. \forall St : t \rightarrow \Omega. \forall l : List\ t. Slist\ t\ l \rightarrow St\ (head\ t\ l)$$

So

$$\mathcal{S}\ s\ [[t]] \equiv s(t)$$

$$\mathcal{S}\ s\ [[\prod t : \kappa. \tau]] \equiv \forall t : \kappa. \forall St : ??? . \mathcal{S}\ \{s, t \mapsto St\}\ [[\tau]]$$

Going up one level

Really, $\mathcal{S}[\tau]$ is similar to $\mathcal{C}[e]$

Just like $\mathcal{C}[e] : \mathcal{S}[\tau]e$ we have $\mathcal{S}[\tau] : \mathcal{S}'[\kappa]\tau$

In a sense, \mathbf{Snat} is treated as “the single value” of $\mathcal{S}'[*]\mathbf{Nat}$

$$\mathcal{S}'[*]\tau \equiv \tau \rightarrow \Omega$$

We also have to define $\mathcal{S}'[\kappa]$ for other kinds

But at least, CC has no kind variables, so the tower stops here

Back to array access

We had

$$\mathit{aref} : \prod t : *, n : \text{Nat}, m : \text{Nat}. (n < m) \rightarrow \text{Array } t \ m \rightarrow t$$

which leads to

$$\begin{aligned} \mathcal{C}[\mathit{aref}] : & \forall t : *, St : t \rightarrow \Omega, n : \text{Nat}, m : \text{Nat}, P : n < m, A : \text{Array } t \ m. \\ & \text{Snat } n \rightarrow \text{Snat } m \rightarrow \text{Slt } n \ m \ P \rightarrow \text{Sarray } t \ m \ A \\ & \rightarrow St (\mathit{aref} \ t \ n \ m \ P \ A) \end{aligned}$$

Requires the usual extra info to make efficient

Conclusion

λ_H is a good target for ML, OO, and dependent types

$\mathcal{S}[\tau]e$ is injective, so all type information is preserved

Traditional compilation techniques can be used,
e.g. CPS and closure conversion become straightforward

We have generalized it to pure type systems

A surface language for λ_H could avoid duplication

Need to extend it to inductive definitions

The type farm has declared independence