On understanding types, data abstraction and effects

Nick Benton

MSR

(with Lennart Beringer, Andrew Kennedy, Martin Hofmann, Gil Hur, Vivek Nigam, Nicolas Tabareau)

Computing Surveys, Vol 17 n. 4, pp 471-522, December 1985

On Understanding Types, Data Abstraction, and Polymorphism

Luca Cardelli
AT&T Bell Laboratories, Murray Hill, NJ 07974
(current address: DEC SRC, 130 Lytton Ave, Palo Alto CA 94301)

Peter Wegner
Dept. of Computer Science, Brown University
Providence, RI 02912

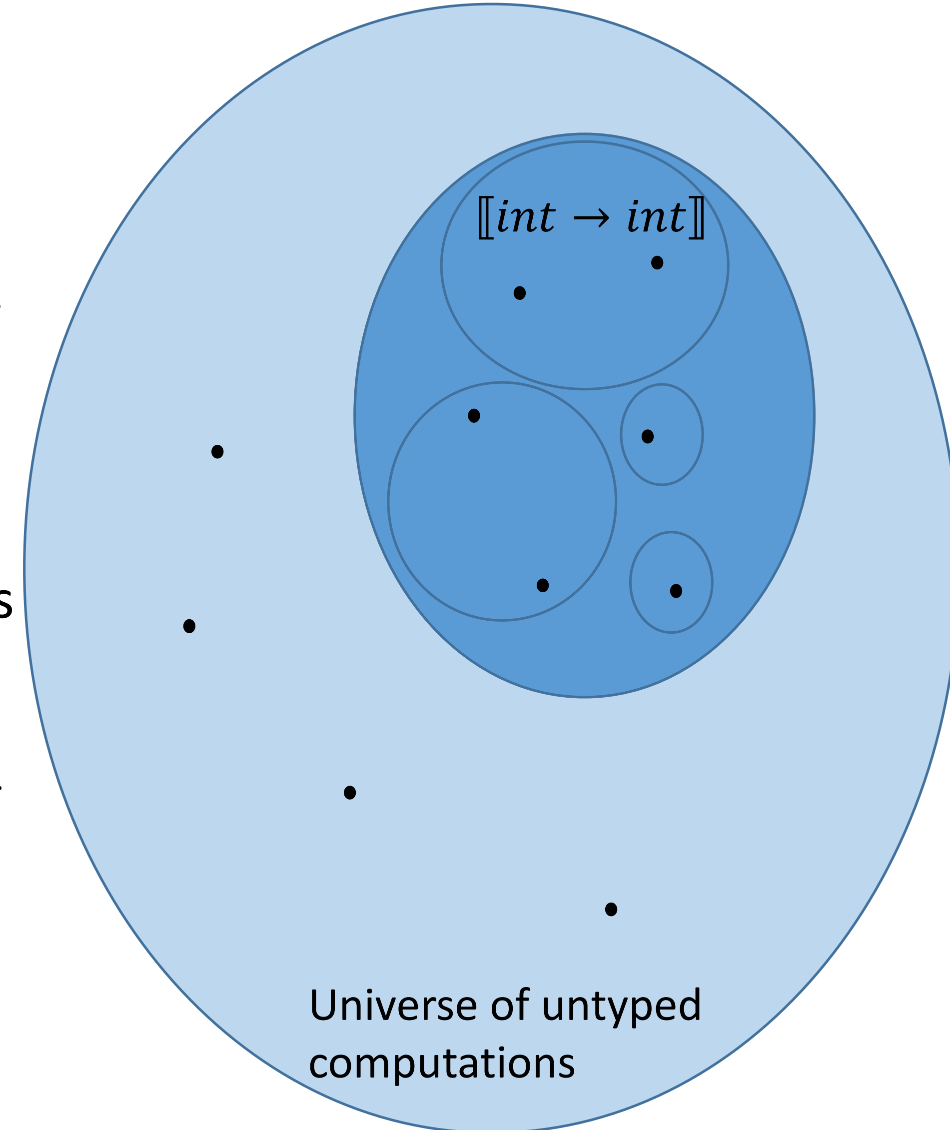
Abstract

Our objective is to understand the notion of *type* in programming languages, present a model of typed, polymorphic programming languages that reflects recent research in type theory, and examine the relevance of recent research to the design of practical programming languages.

Object-oriented languages provide both a framework and a motivation for exploring the interaction among the concepts of type, data abstraction, and polymorphism, since they extend the notion of type to data abstraction and since type inheritance is an important form of polymorphism. We develop a λ-calculus-based model for type systems that allows us to explore these interactions in a simple setting, unencumbered by complexities of production programming languages.

Types as relations

- $\Gamma \vdash M = M' : A$
- $[\![A \to B]\!] = \{(f, f') \mid \forall a\, a', (a, a') \in [\![A]\!] \supset (f a, f' a') \in [\![B]\!]\}$
- $\Gamma \vdash M : A \;\supset ([\![\Gamma \vdash M : A]\!], [\![\Gamma \vdash M : A]\!]) \in [\![\Gamma]\!] \Rightarrow [\![A]\!]$
- Parametric polymorphism (free theorems) understood via quantification over relations beyond equality-like ones
- Original emphasis: finding models for pre-existing type systems for fancy lambda calculi
- Slight shift in viewpoint:
  - Interested in messy, imperative programs and in non-standard type-like properties (for which `pretty' rules are not so obvious)
  - The untyped model is ground truth, relational properties are things we want to establish of our programs. There can be different sound rules for showing programs satisfy properties.
- This talk: outline simple versions of some of the applications we've made of this idea



Relational logic for simple imperative programs

Relational Hoare Logic:

- Move from $\vdash \{P\}C\{Q\}$ to $\vdash C{\sim}C' : \Phi \Rightarrow \Phi'$, where Φ and Φ' are binary relations on states

$$\frac{C{\sim}C' : \Phi \wedge (B\langle 1\rangle \wedge B'\langle 2\rangle) \Rightarrow \Phi' \qquad D{\sim}D' : \Phi \wedge \neg(B\langle 1\rangle \vee B'\langle 2\rangle) \Rightarrow \Phi'}{\text{if } B \text{ then } C \text{ else } D \sim \text{if } B' \text{then } C' \text{else } D' : \Phi \wedge (B\langle 1\rangle = B'\langle 2\rangle) \Rightarrow \Phi'}$$

```
while I<N do        X := Y+1;
  X := Y+1;    ~    while I<N do
  I := I+X;           I := I+X;
```

$: \Phi \Rightarrow \Phi$, where $\Phi = (I\langle 1\rangle = I\langle 2\rangle \wedge N\langle 1\rangle = N\langle 2\rangle \wedge Y\langle 1\rangle = Y\langle 2\rangle)$

- Embeds traditional dataflow analyses, information flow, program slicing *and* gives formally justified program transformations at the same time.

- Further developed for:
  - Crypto protocol verification (CertiCrypt)
  - Differential privacy
  - Relational F*
  - Relational Hoare Type Theory
  - Sel4 verification
  - SymDiff

Formal Parametric Polymorphism

Martín Abadi — Digital Equipment Corporation, Systems Research Center

Luca Cardelli — Digital Equipment Corporation, Systems Research Center

Pierre-Louis Curien — CNRS, Ecole Normale Supérieure

Abstract

A polymorphic function is parametric if its behavior does not depend on the type at which it is instantiated. Starting with Reynolds's work, the study of parametricity is typically semantic. In this

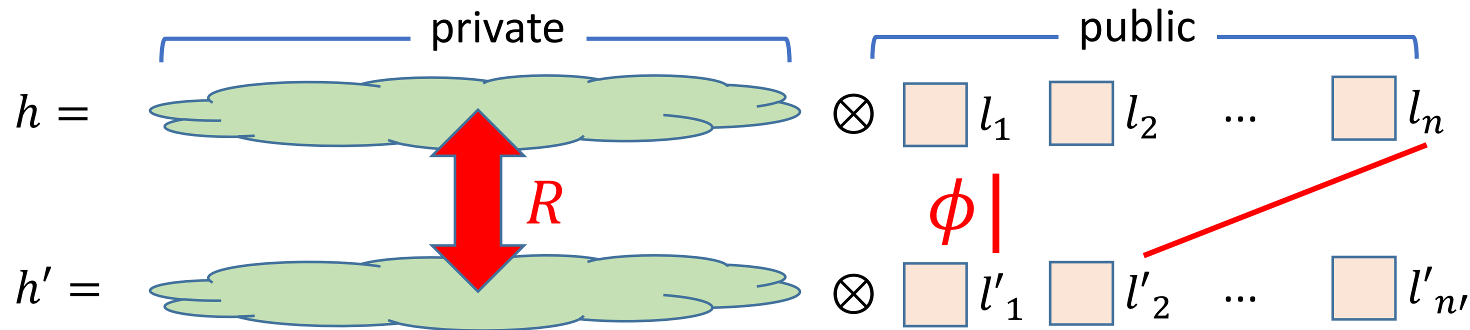
Models of ML-like languages

- Subtle combination of higher-order functions and dynamically allocated references

```
let x = ref 0 in
  fn () => x := !x-1; -!x

        ≈

 let x = ref 0 in
   fn () => x := !x+1; !x
```



Kripke logical relation. World is $R\phi$ where R a relation on heaps and ϕ a partial bijection on locations

- $H(R\phi) = \{(h, h') \mid h = h_1 \uplus h_2, h' = h'_1 \uplus h'_2, (h_1, h'_1) \in R \ \wedge\ \forall (l, l') \in \phi, h_2(l) = h'_2(l')\}$
- $[\![\mathrm{int}]\!](R\phi) = \{(n, n) | n \in \mathbb{Z}\}$
- $[\![\mathrm{ref}]\!](R\phi) = \phi$
- $[\![A \to B]\!](R\phi) =$
  $\{(f, f') | \forall \overline{R\phi} \trianglerighteq R\phi, (h, h') \in H(\overline{R\phi}), (a, a') \in [\![A]\!](\overline{R\phi}), \quad \exists \overline{\overline{R\phi}} \trianglerighteq \overline{R\phi}, (f\ h\ a, f'h'a') \in H\left(\overline{\overline{R\phi}}\right) \times [\![B]\!](\overline{\overline{R\phi}})\}$

- Fundamental theorem: $\Gamma \vdash M{:}\,A \ \supset ([\![\Gamma \vdash M{:}\,A]\!], [\![\Gamma \vdash M{:}\,A]\!]) \in ([\![\Gamma]\!] \Rightarrow [\![A]\!])(R\phi)$
- Adequacy: $([\![\Gamma \vdash M{:}\,A]\!], [\![\Gamma \vdash N{:}\,A]\!]) \in ([\![\Gamma]\!] \Rightarrow [\![A]\!])(.) \ \supset \Gamma \vDash M \cong N{:}\,A$

Semantics of effect systems

- Refined type systems tracking an upper bound on side-effects

$$\frac{\Theta \vdash V : X}{\Theta \vdash val\ V : T_{\emptyset} X} \qquad \frac{\Theta \vdash M : T_{\epsilon_1} X \quad \Theta, x : X \vdash N : T_{\epsilon_2} Y}{\Theta \vdash let\ x \Leftarrow M\ in\ N : T_{\epsilon_1 \cup \epsilon_2} Y} \qquad \frac{}{\Theta \vdash !\,\ell : T_{\{r_\ell\}} int} \qquad \frac{\Theta \vdash V : int}{\Theta \vdash \ell := V : T_{\{w_\ell\}} unit}$$

- Preserving all relations preserved by all the operations

$$[\![T_\epsilon X]\!] \subseteq (S \to S \times [\![UX]\!]) \times (S \to S \times [\![UX]\!])$$

$$[\![T_\epsilon X]\!] = \bigcap_{R \in \mathcal{R}_\epsilon} R \Rightarrow R \times [\![X]\!]$$

$$\mathcal{R}_\epsilon, \mathcal{R}_e \subseteq \mathbb{P}(S \times S)$$

$$\mathcal{R}_\epsilon = \bigcap_{e \in \epsilon} \mathcal{R}_e$$

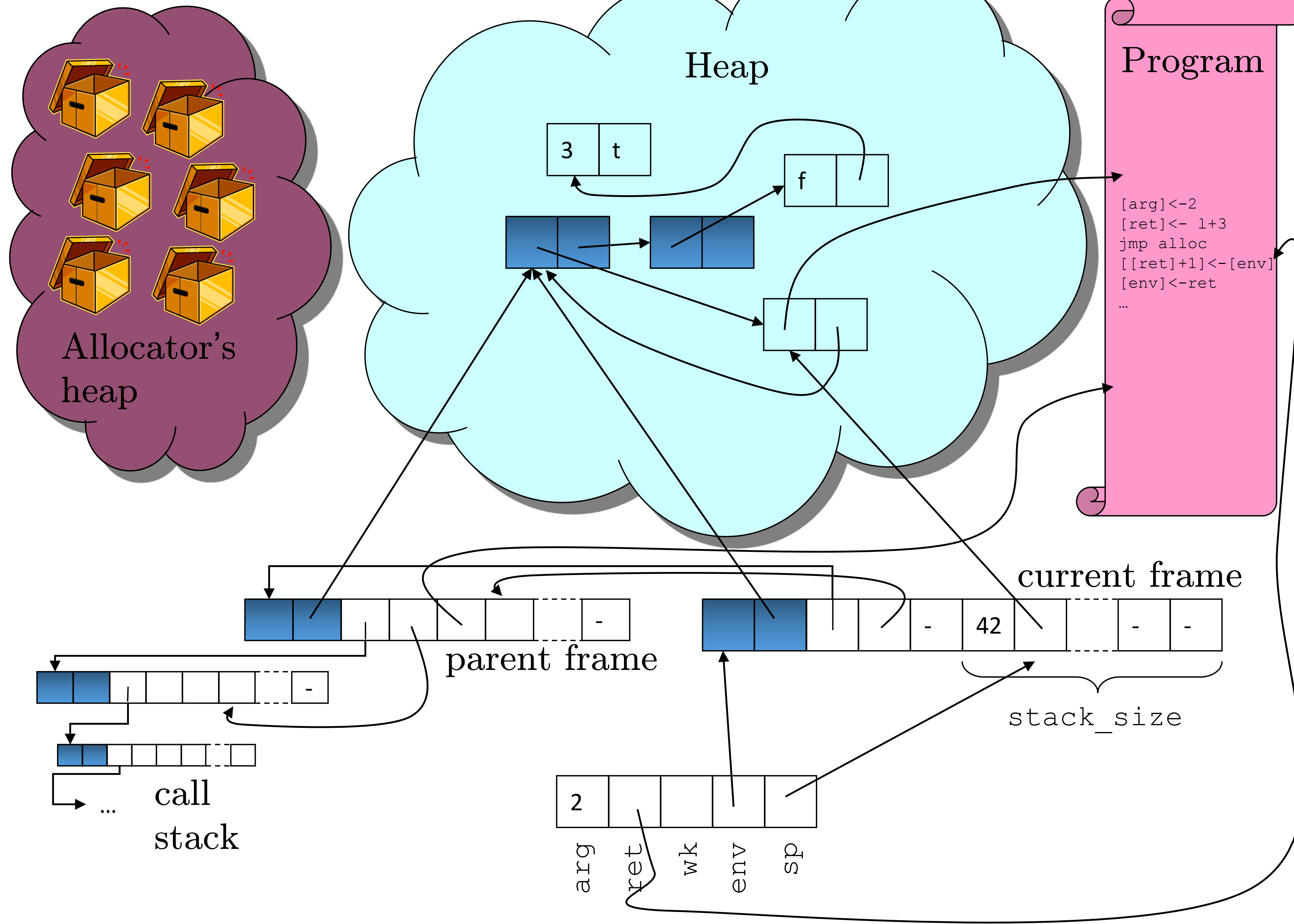
$$\mathcal{R}_{r_\ell} = \{R \mid \forall (s, s') \in R, s\,\ell = s'\ell\}$$

$$\mathcal{R}_{w_\ell} = \{R \mid \forall (s, s') \in R, n \in \mathbb{Z}. (s[\ell \mapsto n], s'[\ell \mapsto n]) \in R\}$$

$$\frac{\Theta \vdash M : T_{\epsilon_1} X \quad \Theta, x : X, y : X \vdash N : T_{\epsilon_2} Y}{\Theta \vdash let\ x \Leftarrow M; y \Leftarrow M\ in\ N = let\ x \Leftarrow M\ in\ N[x/y] : T_{\epsilon_1 \cup \epsilon_2} Y}\ rds(\epsilon_1) \cap wrs(\epsilon_1) = \emptyset$$

Semantic type soundness for a compiler

- Simple functional language compiled to idealized assembly code
- Source types interpreted as binary relations over low level machine
  - Only talk about observable behaviour
  - Define calling conventions but independent of particular compiler
- Compositional proof that the code produced from source phrase of type A is related to itself by $[\![A]\!]$
  - Uses a relational separation logic for assembly code (inc. code pointers)
  - Links to modular proof of a memory allocator
- Ensures functions give (suitably) equivalent results when given equivalent arguments
  - Independent of how they're laid out in memory, what the code in closures is, what the allocator does, etc.
- Precise behavioural contract for safely linking foreign code



Allocator's heap
Heap
3
t
f
Program
[arg]<-2
[ret]<- 1+3
jmp alloc
[[ret]+1]<-[env]
[env]<-ret
…
current frame
-
42
-
-
parent frame
-
-
stack_size
…
call stack
2
arg
ret
wk
env
sp

Two other applications

- Compositional compiler correctness
  - Full functional correctness for functional compilers
  - Using relations between high level and low level (and biorthogonality)
- Abstract effects
  - Instead of tracking effects at the level of individual locations, work at the level of whole abstract datastructures
  - E.g. set lookup operation assigned a read-but-not-write effect, even if it does internal, non-observable, writes to rebalance a datastructure
  - Theory involves relational account of abstract locations and an exciting new kind of "proof relevant" logical relation for interpreting types

Summary

- Relational approach is powerful, elegant and effective at taming side effects
- Note that there's no instrumentation, or talk of going wrong
- "All type systems are about information flow"
- "Preserving all relations preserved by all the operations"
- Semantics first
  - Syntactic types are a convenient interface language for components
  - Can show that a component inhabits a type by simple syntactic or complex semantic techniques

Happy Birthday, Luca!