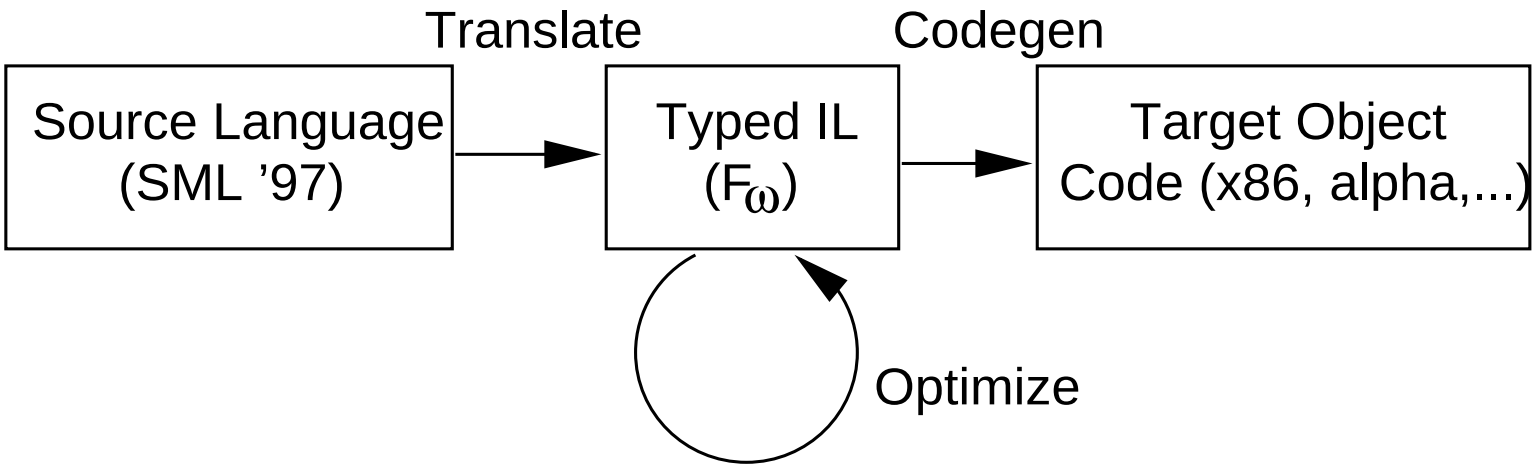
Implementing Typed Intermediate Languages

Zhong Shao
Christopher League
Stefan Monnier

Yale University

ICFP'98, Baltimore
29 September 1998

Structure of a type-preserving compiler



Advantages of type-preserving compilation

- Representation analysis
- Tagless GC
- Intensional type analysis
- Type-directed partial evaluation
- Secure mobile code (PCC, TAL)

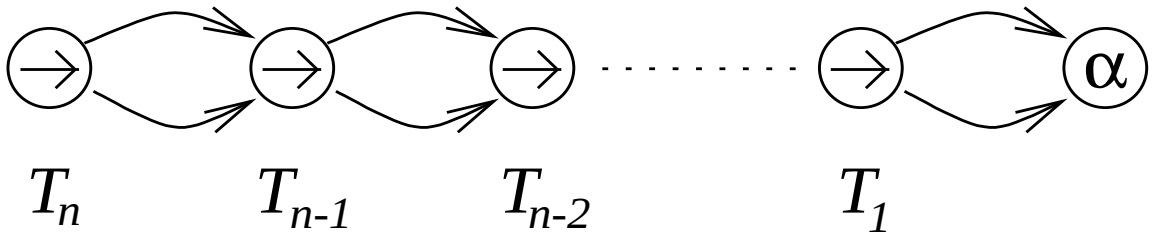
Why are these not yet used in production systems?

- 'Production systems' are rare in FP community
- Hard to scale these techniques to compile *large* programs

Types can be enormous

```
fun f x = x
fun toy() =
   let fun g y = ((((f f) f) f) ... f) y
    in g 3
   end
```

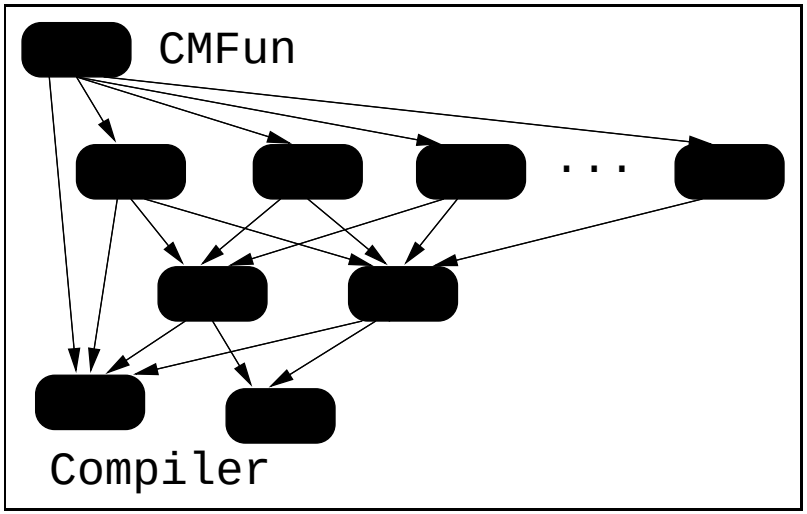
Rightmost `f`: $T_1 = \alpha \to \alpha$. Leftmost `f`: $T_n = T_{n-1} \to T_{n-1}$



Real programs have large types

`cm.sml` contains:

100 lines ML source code
36 functor applications
80 structure references



CM compilation time, untyped SML/NJ: 2-3 minutes
(initial version) typed SML/NJ: 12 minutes

Our contribution

- *Scalable* type-preserving production compiler
- Combination of techniques for handling large types
  - de Bruijn indices
  - Lazy reduction using explicit substitutions
  - Hash-consing plus memoization
- Demonstrated in SML/NJ since January 1997

Criteria for efficient implementation of a typed IL

- Compact space usage *(guaranteed)*
- Linear manipulation
- Cheap equivalence test
- Clean interface

FLINT Typed Intermediate Language

$$
\begin{aligned}
\kappa &::= \Omega \mid \kappa_1 \rightarrow \kappa_2 \\
\mu &::= t \mid \texttt{Int} \mid \mu_1 \rightarrow \mu_2 \mid \lambda t :: \kappa . \mu \mid \mu_1[\mu_2] \\
\sigma &::= T(\mu) \mid \sigma_1 \rightarrow \sigma_2 \mid \forall t :: \kappa . \sigma \\
e &::= i \mid x \mid \lambda x : \sigma . e \mid @x_1\ x_2 \mid \Lambda t :: \kappa . e \mid x[\mu] \\
&\quad \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2
\end{aligned}
$$

Simple interface to type language

```
signature FLINTTYPE = sig
    (* abstract types *)
    type tkind
    type tyc
    (* constructors *)
    val tcc_int    : tyc
    val tcc_var    : tvar -> tyc
    val tcc_arrow  : tyc * tyc -> tyc
    val tcc_fn     : tkind * tyc -> tyc
    val tcc_app    : tyc * tyc -> tyc
    (* selectors *)
    val tcd_arrow  : tyc -> tyc * tyc
       :
    (* predicates *)
    val tcp_arrow  : tyc -> bool
       :
       :
end
```

Client can treat each type as its intension

```
val pr = tcc_fn ...                          (* λt::Ω . ×(t,t) *)
val t1 = tcc_app (pr, tcc_int)               (* (λt::Ω . ×(t,t))[Int] *)
val t2 = tcc_tuple [tcc_int, tcc_int]        (* ×(Int,Int) *)

tcp_app t1        ⇒  false
tcp_tuple t1      ⇒  true
tc_eq (t1, t2)    ⇒  true
```

Reduction, however, is lazy.

Use de Bruijn indices to represent type variables

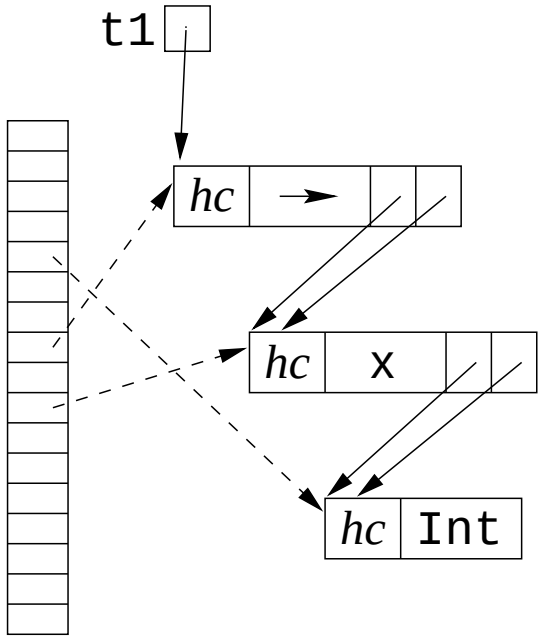
$$\underline{\lambda\alpha :: \Omega}. \cdots \underline{\alpha} \cdots$$
$$\quad \lambda\beta :: \Omega. \cdots \underline{\alpha} \times \beta \cdots$$

$$\Downarrow$$

$$\underline{\lambda\Omega}. \cdots \underline{\#1} \cdots$$
$$\quad \lambda\Omega. \cdots \underline{\#2} \times \#1 \cdots$$

Hash-cons all nodes into a global table

```
val t1 = tcc_arrow(tcc_tuple [tcc_int, tcc_int],
                   tcc_tuple [tcc_int, tcc_int])
```



Suspension terms support lazy reduction

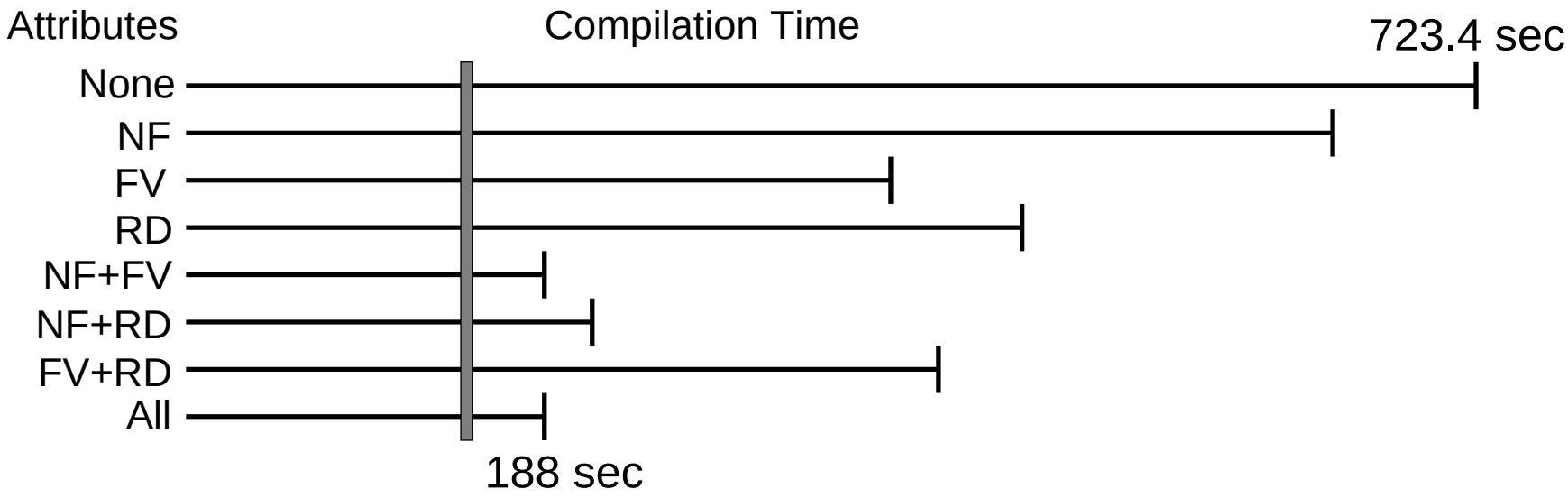
$\texttt{Env}(\mu, \rho)$ where μ is type with free variables
ρ is a substitution (encoded)

$$
\begin{aligned}
(\lambda\kappa.\#1 \times \#1)[\mu] &\Rightarrow \texttt{Env}(\#1 \times \#1,\ \rho) \\
&\Rightarrow \texttt{Env}(\#1, \rho) \times \texttt{Env}(\#1, \rho) \\
&\Rightarrow \rho(\#1) \times \rho(\#1) \\
&\Rightarrow \mu \times \mu
\end{aligned}
$$

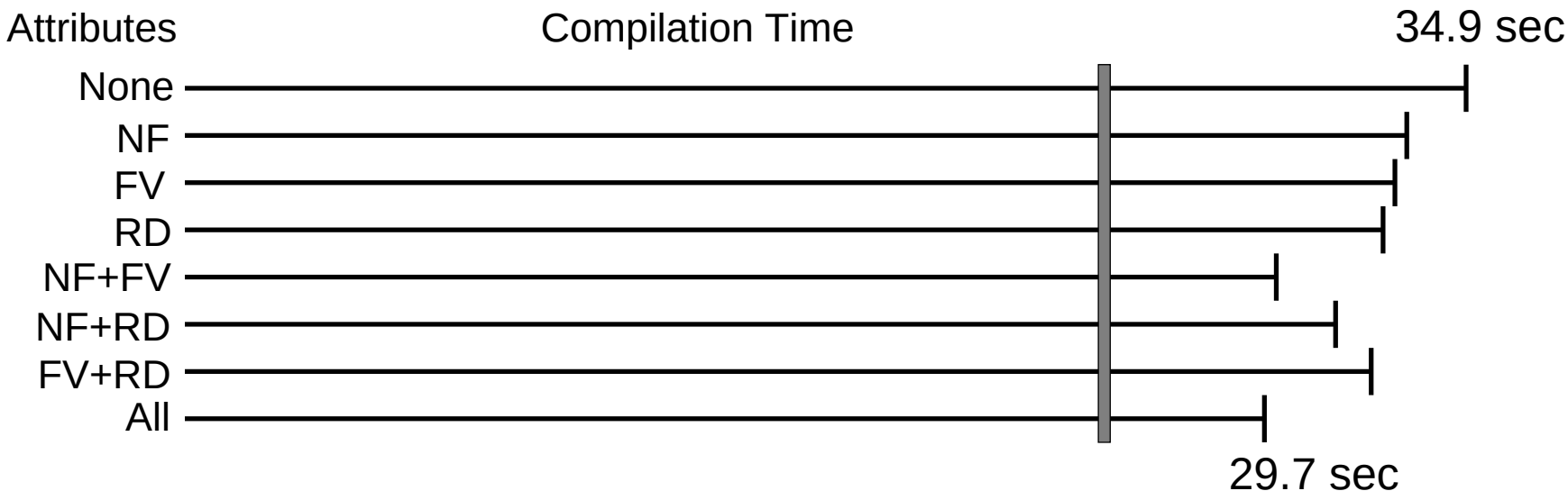
Memoize useful information about each node

- Normal form flag
- Set of free type variables (encoded)
- Reduction results

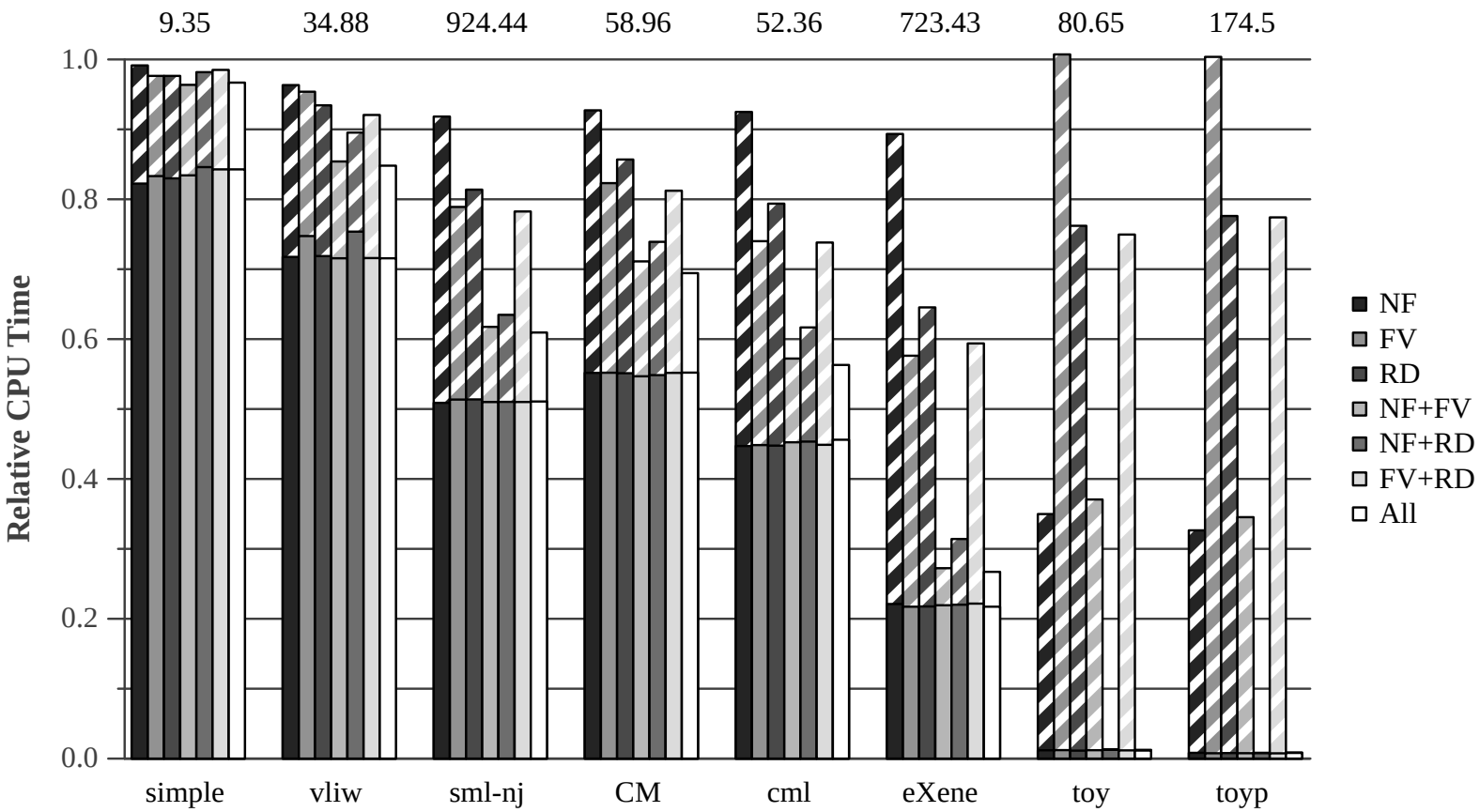
Memoization benefit for eXene



Memoization benefit for VLIW



Memoization benefits



Problem: de Bruijn indices exposed in term language

Type annotations might contain de Bruijn indices:

$$\Lambda\underline{\Omega}\ .\ \lambda x : \underline{\#1}\ .\ x$$

Must adjust indices when, e.g., inlining:

$$\Lambda\underline{\Omega}.\texttt{let}\ f = \underbrace{\lambda x : \underline{\#1}.x}$$
$$\texttt{in}\ \Lambda\Omega.\cdots f\ y\cdots$$

Solution: use named variables for type abstraction

$$(\Lambda\underline{\alpha :: \Omega}.\lambda x : \underline{\alpha}.x) : \forall\Omega.\#1 \rightarrow \#1$$

- *Free* variables in type annotations are all named
- Convert to indices when building $\forall$ type
- Low cost (*usually* $\leq 1\%$ increase in compilation time)

Alternative approach: lettype

Advantages:

- Bottom-up representation
- Sharing is explicit

$$\begin{array}{l}\texttt{lettype}\, t_1 = \texttt{Int} \times \texttt{Int} \\ \texttt{in lettype}\, t_2 = t_1 \to t_1 \\ \quad \texttt{in}\ (\lambda x{:}t_1.x) : t_2\end{array}$$

Disadvantages:

- No compact normal form
- Equivalence test is expensive
- Need to *find* redundancy, no guarantee
- Unclear how to provide clean interface

Contribution

- First scalable type-preserving production compiler
- Combination of techniques for handling large types
- Demonstrated in SML/NJ since January 1997