A Rewriting Logic Approach to Type Inference

Chucky Ellison, Traian Florin Șerbănuță, and Grigore Roșu

University of Illinois at Urbana-Champaign

June 14, 2008

Outline

What We Have Done

- K, a rewriting-logic inspired framework for language development
- Shown how the K can also encompass type systems
  - Works on both imperative and functional languages
  - Can define as both type checkers and inferencers
  - Executable!
  - Formally analyzable – proofs of soundness for type systems

What We Are Going to Talk About

- Short overview of K framework
- How we use K to define Milner's Type Inferencer $\mathcal{W}$
- Proof techniques developed to analyze the inferencer
  - Morphism from language configurations to type configurations
  - Abstract type system

Introducing K: Rules

- Structural rules (reversible transitions, heating/cooling rules):
  $LHS = RHS$, or $LHS \rightleftharpoons RHS$;
- Semantic rules (configuration-modifying transitions):
  $LHS \longrightarrow RHS$.
- Contextual rewriting style:
  Use $C[\underset{R_1}{\underline{L_1}}, \ldots, \underset{R_N}{\underline{L_N}}]$ instead of $C[L_1, \ldots, L_N] \longrightarrow C[R_1, \ldots, R_N]$
- List and set comprehension
  - Match middle – $\langle\!| X |\!\rangle$, prefix – $(\!| X |\!\rangle$, suffix – $\langle\!| X |\!)$
  - Works well with contextual rewriting. E.g., stating idempotency: $\langle\!| X \underset{\cdot}{\underline{X}} |\!\rangle$, where $\cdot$ means the identity for sets.

Outline

Exp Syntax

$$
\begin{array}{lcll}
\mathit{Var} & ::= & \text{standard identifiers} & \\
\mathit{Exp} & ::= & \mathit{Var} \mid \ldots \text{ add basic values (Bools, ints, etc.)} & \\
& \mid & \lambda\, \mathit{Var} \,.\, \mathit{Exp} & \\
& \mid & \mathit{Exp}\ \mathit{Exp} & [\mathit{strict}] \\
& \mid & \mu\, \mathit{Var} \,.\, \mathit{Exp} & \\
& \mid & \text{if } \mathit{Exp} \text{ then } \mathit{Exp} \text{ else } \mathit{Exp} & [\mathit{strict}(1)] \\
& \mid & \text{let } \mathit{Var} = \mathit{Exp} \text{ in } \mathit{Exp} & [\mathit{strict}(2)] \\
& \mid & \text{letrec } \mathit{Var}\ \mathit{Var} = \mathit{Exp} \text{ in } \mathit{Exp} & \\
& & & [\text{letrec } f\ x = e \text{ in } e' \leftrightharpoons \text{let } = f \mu f.(\lambda x.e) \text{ in } e']
\end{array}
$$

Desugaring Strictness Rules

- Strictness attributes on language constructs . . .

$$
\begin{array}{lcll}
\mathit{Exp} & ::= & \mathit{Exp}\ \mathit{Exp} & [\mathit{strict}] \\
 & \mid & \text{if } \mathit{Exp} \text{ then } \mathit{Exp} \text{ else } \mathit{Exp} & [\mathit{strict}(1)] \\
 & \mid & \text{let } \mathit{Var} = \mathit{Exp} \text{ in } \mathit{Exp} & [\mathit{strict}(2)]
\end{array}
$$

- . . . are desugared into “evaluation” operations . . .

$$k_1\ k_2 = k_1 \curvearrowright \square\ k_2$$
$$k_1\ k_2 = k_2 \curvearrowright k_1\ \square$$
$$\text{if } k \text{ then } k_1 \text{ else } k_2 = k \curvearrowright \text{if } \square \text{ then } k_1 \text{ else } k_2$$
$$\text{let } \mathit{Var} = k \text{ in } k_1 = k \curvearrowright \text{let } \mathit{Var} = \square \text{ in } k_1$$

- . . . by using “□”-based constructs to freeze computations.

Exp Configuration and Semantics

$$\begin{array}{rcl} \mathit{Val} & ::= & \lambda\, \mathit{Var}\,.\, \mathit{Exp} \mid \ldots(\text{Bools, ints, etc.}) \\ \mathit{Result} & ::= & \mathit{Val} \\ \mathit{KProper} & ::= & \mu\, \mathit{Var}\,.\, \mathit{Exp} \\ \mathit{ConfigItem} & ::= & (\!|K|\!)_k \\ \mathit{Config} & ::= & \mathit{Val} \mid [\![\mathit{Exp}]\!] \mid \mathit{Set}[\mathit{ConfigItem}] \end{array}$$

$$(\!|\, \frac{(\lambda x.e)\; v}{e[x \leftarrow v]} \,|\!)_k \qquad (\!|\, \frac{\mu\, x.e}{e[x \leftarrow \mu\, x.e]} \,|\!)_k$$

$$\text{if true then } e_1 \text{ else } e_2 \rightarrow e_1 \qquad \text{if false then } e_1 \text{ else } e_2 \rightarrow e_2$$

Let Polymorphism

- The following would not work without let polymorphism:
  `let` $f = \lambda x.x$ `in if` f `true then` f `else` $(\lambda x.1)$
- Why? Type of f is constrained to both $bool \longrightarrow bool$ and $t \longrightarrow int$
- Solution:
  - when typing f, make it parametric in unbounded type variables
  - instantiate them with fresh ones whenever f is later used

  Thus obtained type of above expression is $int \longrightarrow int$
- Notice: expression evaluates to f, which is polymorphic, thus more general than inferred type

$\mathcal{W}$ Inferencer Syntax

$$
\begin{array}{rcll}
\textit{Var} & ::= & \text{standard identifiers} & \\
\textit{Exp} & ::= & \textit{Var} \mid \ldots \text{ add basic values (Bools, ints, etc.)} & \\
& \mid & \lambda\, \textit{Var} \,.\, \textit{Exp} & \\
& \mid & \textit{Exp}\ \textit{Exp} & [\textit{strict}] \\
& \mid & \mu\, \textit{Var} \,.\, \textit{Exp} & \\
& \mid & \text{if } \textit{Exp} \text{ then } \textit{Exp} \text{ else } \textit{Exp} & [\textit{strict}] \\
& \mid & \text{let } \textit{Var} = \textit{Exp} \text{ in } \textit{Exp} & [\textit{strict}(2)] \\
& \mid & \text{letrec } \textit{Var}\ \textit{Var} = \textit{Exp} \text{ in } \textit{Exp} & \\
& & & [\text{letrec } f\ x = e \text{ in } e' \leftrightharpoons \text{let } f = \mu f.(\lambda x.e) \text{ in } e']
\end{array}
$$

$\mathcal{W}$ Inferencer Configuration Syntax

$$
\begin{array}{rcl}
K & ::= & \cdots \mid \mathit{Type} \rightarrow K \quad [\mathit{strict}(2)] \\
\mathit{Result} & ::= & \mathit{Type} \\
\mathit{TEnv} & ::= & \mathit{Map}[\mathit{Name}, \mathit{Type}] \\
\mathit{Type} & ::= & \ldots \mid \mathit{let}(\mathit{Type}) \\
\mathit{ConfigItem} & ::= & (\!|K|\!)_{k} \mid (\!|\mathit{TEnv}|\!)_{\mathit{tenv}} \mid (\!|\mathit{Eqns}|\!)_{\mathit{eqns}} \mid (\!|\mathit{TypeVar}|\!)_{\mathit{nextType}} \\
\mathit{Config} & ::= & \mathit{Type} \mid [\![K]\!] \mid [\![\mathit{Set}[\mathit{ConfigItem}]]\!] \\
\mathit{Type} & ::= & \ldots \mid \mathit{int} \mid \mathit{bool} \mid \mathit{Type} \mapsto \mathit{Type} \mid \mathit{TypeVar} \\
\mathit{Eqn} & ::= & \mathit{Type} \equiv \mathit{Type} \\
\mathit{Eqns} & ::= & \mathit{Set}[\mathit{Eqn}]
\end{array}
$$

$\mathcal{W}$ by Example

- $\llparenthesis \text{let } f = \lambda x.x \text{ in if } f \text{ true then } f \text{ else } \lambda x.1 \rrparenthesis_k \llparenthesis \cdot \rrparenthesis_\Gamma \llparenthesis \cdot \rrparenthesis_\mathcal{E}$
- $\llparenthesis \text{if } f \text{ true then } f \text{ else } \lambda x.1 \rrparenthesis_k \llparenthesis f = let(t \to t) \rrparenthesis_\Gamma \llparenthesis \cdot \rrparenthesis_\mathcal{E}$
- $\llparenthesis \text{if } t_1 \text{ then } f \text{ else } \lambda x.1 \rrparenthesis_k \llparenthesis f = let(t \to t) \rrparenthesis_\Gamma \llparenthesis t_1 = bool \rrparenthesis_\mathcal{E}$
- $\llparenthesis \text{if } t_1 \text{ then } t_2 \to t_2 \text{ else } \lambda x.1 \rrparenthesis_k \llparenthesis f = let(t \to t) \rrparenthesis_\Gamma \llparenthesis t_1 = bool \rrparenthesis_\mathcal{E}$
- $\llparenthesis \text{if } t_1 \text{ then } t_2 \to t_2 \text{ else } t_3 \to int \rrparenthesis_k \llparenthesis f = let(t \to t) \rrparenthesis_\Gamma \llparenthesis t_1 = bool \rrparenthesis_\mathcal{E}$
- $\llparenthesis t_2 \to t_2 \rrparenthesis_k \llparenthesis f = let(t \to t) \rrparenthesis_\Gamma \llparenthesis t_1 = bool, t_2 \to t_2 = t_3 \to int \rrparenthesis_\mathcal{E}$
- $\llparenthesis t_2 \to t_2 \rrparenthesis_k \llparenthesis f = let(t \to t) \rrparenthesis_\Gamma \llparenthesis t_1 = bool, t_2 = t_3, t_2 = int \rrparenthesis_\mathcal{E}$
- $int \to int$

Let us exemplify $\mathcal{W}$ by typing expression above

$\mathcal{W}$ by Example

- $\llparenthesis \text{let } f = \lambda x.x \text{ in if } f \text{ true then } f \text{ else } \lambda x.1 \rrparenthesis_k \llparenthesis \cdot \rrparenthesis_\Gamma \llparenthesis \cdot \rrparenthesis_\mathcal{E}$
- $\llparenthesis \text{if } f \text{ true then } f \text{ else } \lambda x.1 \rrparenthesis_k \llparenthesis f = let(t \to t) \rrparenthesis_\Gamma \llparenthesis \cdot \rrparenthesis_\mathcal{E}$
- $\llparenthesis \text{if } t_1 \text{ then } f \text{ else } \lambda x.1 \rrparenthesis_k \llparenthesis f = let(t \to t) \rrparenthesis_\Gamma \llparenthesis t_1 = bool \rrparenthesis_\mathcal{E}$
- $\llparenthesis \text{if } t_1 \text{ then } t_2 \to t_2 \text{ else } \lambda x.1 \rrparenthesis_k \llparenthesis f = let(t \to t) \rrparenthesis_\Gamma \llparenthesis t_1 = bool \rrparenthesis_\mathcal{E}$
- $\llparenthesis \text{if } t_1 \text{ then } t_2 \to t_2 \text{ else } t_3 \to int \rrparenthesis_k \llparenthesis f = let(t \to t) \rrparenthesis_\Gamma \llparenthesis t_1 = bool \rrparenthesis_\mathcal{E}$
- $\llparenthesis t_2 \to t_2 \rrparenthesis_k \llparenthesis f = let(t \to t) \rrparenthesis_\Gamma \llparenthesis t_1 = bool, t_2 \to t_2 = t_3 \to int \rrparenthesis_\mathcal{E}$
- $\llparenthesis t_2 \to t_2 \rrparenthesis_k \llparenthesis f = let(t \to t) \rrparenthesis_\Gamma \llparenthesis t_1 = bool, t_2 = t_3, t_2 = int \rrparenthesis_\mathcal{E}$
- $int \to int$

Type $\lambda x.x$: bind x to a new type variable t and obtain $t \to t$
Bind f to special type $let(t \to t)$ and begin typing the body

$\mathcal{W}$ by Example

- $(\!|\text{let } f = \lambda x.x \text{ in if } f \text{ true then } f \text{ else } \lambda x.1|\!)_k (\!|\cdot|\!)_\Gamma (\!|\cdot|\!)_\mathcal{E}$
- $(\!|\text{if } f \text{ true then } f \text{ else } \lambda x.1|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|\cdot|\!)_\mathcal{E}$
- $(\!|\text{if } t_1 \text{ then } f \text{ else } \lambda x.1|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool|\!)_\mathcal{E}$
- $(\!|\text{if } t_1 \text{ then } t_2 \to t_2 \text{ else } \lambda x.1|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool|\!)_\mathcal{E}$
- $(\!|\text{if } t_1 \text{ then } t_2 \to t_2 \text{ else } t_3 \to int|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool|\!)_\mathcal{E}$
- $(\!|t_2 \to t_2|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool, t_2 \to t_2 = t_3 \to int|\!)_\mathcal{E}$
- $(\!|t_2 \to t_2|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool, t_2 = t_3, t_2 = int|\!)_\mathcal{E}$
- $int \to int$

Get a fresh instance of f, $t_1 \to t_1$. Type f `true` to t_1.
Add constraint $t_1 = bool$

$\mathcal{W}$ by Example

- $(\!|\text{let } f = \lambda x.x \text{ in if } f \text{ true then } f \text{ else } \lambda x.1|\!)_k (\!|\cdot|\!)_\Gamma (\!|\cdot|\!)_\mathcal{E}$
- $(\!|\text{if } f \text{ true then } f \text{ else } \lambda x.1|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|\cdot|\!)_\mathcal{E}$
- $(\!|\text{if } t_1 \text{ then } f \text{ else } \lambda x.1|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool|\!)_\mathcal{E}$
- $(\!|\text{if } t_1 \text{ then } t_2 \to t_2 \text{ else } \lambda x.1|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool|\!)_\mathcal{E}$
- $(\!|\text{if } t_1 \text{ then } t_2 \to t_2 \text{ else } t_3 \to int|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool|\!)_\mathcal{E}$
- $(\!|t_2 \to t_2|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool, t_2 \to t_2 = t_3 \to int|\!)_\mathcal{E}$
- $(\!|t_2 \to t_2|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool, t_2 = t_3, t_2 = int|\!)_\mathcal{E}$
- $int \to int$

Type f: Get a fresh instance of f, $t_2 \to t_2$

$\mathcal{W}$ by Example

- $(\!|\text{let } f = \lambda x.x \text{ in if } f \text{ true then } f \text{ else } \lambda x.1|\!)_k (\!|\cdot|\!)_\Gamma (\!|\cdot|\!)_\mathcal{E}$
- $(\!|\text{if } f \text{ true then } f \text{ else } \lambda x.1|\!)_k (\!|f = let(t \longrightarrow t)|\!)_\Gamma (\!|\cdot|\!)_\mathcal{E}$
- $(\!|\text{if } t_1 \text{ then } f \text{ else } \lambda x.1|\!)_k (\!|f = let(t \longrightarrow t)|\!)_\Gamma (\!|t_1 = bool|\!)_\mathcal{E}$
- $(\!|\text{if } t_1 \text{ then } t_2 \longrightarrow t_2 \text{ else } \lambda x.1|\!)_k (\!|f = let(t \longrightarrow t)|\!)_\Gamma (\!|t_1 = bool|\!)_\mathcal{E}$
- $(\!|\text{if } t_1 \text{ then } t_2 \longrightarrow t_2 \text{ else } t_3 \longrightarrow int|\!)_k (\!|f = let(t \longrightarrow t)|\!)_\Gamma (\!|t_1 = bool|\!)_\mathcal{E}$
- $(\!|t_2 \longrightarrow t_2|\!)_k (\!|f = let(t \longrightarrow t)|\!)_\Gamma (\!|t_1 = bool, t_2 \longrightarrow t_2 = t_3 \longrightarrow int|\!)_\mathcal{E}$
- $(\!|t_2 \longrightarrow t_2|\!)_k (\!|f = let(t \longrightarrow t)|\!)_\Gamma (\!|t_1 = bool, t_2 = t_3, t_2 = int|\!)_\mathcal{E}$
- $int \longrightarrow int$

Type $\lambda x.1$: bind x to new type var t_3; conclude with $t_3 \longrightarrow int$

$\mathcal{W}$ by Example

- $(\!|\text{let } f = \lambda x.x \text{ in if } f \text{ true then } f \text{ else } \lambda x.1|\!)_k (\!|\cdot|\!)_\Gamma (\!|\cdot|\!)_\mathcal{E}$
- $(\!|\text{if } f \text{ true then } f \text{ else } \lambda x.1|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|\cdot|\!)_\mathcal{E}$
- $(\!|\text{if } t_1 \text{ then } f \text{ else } \lambda x.1|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool|\!)_\mathcal{E}$
- $(\!|\text{if } t_1 \text{ then } t_2 \to t_2 \text{ else } \lambda x.1|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool|\!)_\mathcal{E}$
- $(\!|\text{if } t_1 \text{ then } t_2 \to t_2 \text{ else } t_3 \to int|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool|\!)_\mathcal{E}$
- $(\!|t_2 \to t_2|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool, t_2 \to t_2 = t_3 \to int|\!)_\mathcal{E}$
- $(\!|t_2 \to t_2|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool, t_2 = t_3, t_2 = int|\!)_\mathcal{E}$
- $int \to int$

Type if t_1 then $t_2 \to t_2$ else $t_3 \to int$ to $t_2 \to t_2$.
Add constraints $t_1 = bool$ (already there) and $t_2 \to t_2 = t_3 \to int$.

$\mathcal{W}$ by Example

- $(\!|$let $f = \lambda x.x$ in if f true then f else $\lambda x.1|\!)_k (\!|\cdot|\!)_\Gamma (\!|\cdot|\!)_\mathcal{E}$
- $(\!|$if f true then f else $\lambda x.1|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|\cdot|\!)_\mathcal{E}$
- $(\!|$if t_1 then f else $\lambda x.1|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool|\!)_\mathcal{E}$
- $(\!|$if t_1 then $t_2 \to t_2$ else $\lambda x.1|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool|\!)_\mathcal{E}$
- $(\!|$if t_1 then $t_2 \to t_2$ else $t_3 \to int|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool|\!)_\mathcal{E}$
- $(\!|t_2 \to t_2|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool, t_2 \to t_2 = t_3 \to int|\!)_\mathcal{E}$
- $(\!|t_2 \to t_2|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool, t_2 = t_3, t_2 = int|\!)_\mathcal{E}$
- $int \to int$

Constraints solvable: $t_1 = bool$, $t_2 = t_3 = int$.

Final type: $int \to int$.

$\mathcal{W}$ by Example

- $(\!|\text{let } f = \lambda x.x \text{ in if } f \text{ true then } f \text{ else } \lambda x.1|\!)_k (\!|\cdot|\!)_\Gamma (\!|\cdot|\!)_\mathcal{E}$
- $(\!|\text{if } f \text{ true then } f \text{ else } \lambda x.1|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|\cdot|\!)_\mathcal{E}$
- $(\!|\text{if } t_1 \text{ then } f \text{ else } \lambda x.1|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool|\!)_\mathcal{E}$
- $(\!|\text{if } t_1 \text{ then } t_2 \to t_2 \text{ else } \lambda x.1|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool|\!)_\mathcal{E}$
- $(\!|\text{if } t_1 \text{ then } t_2 \to t_2 \text{ else } t_3 \to int|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool|\!)_\mathcal{E}$
- $(\!|t_2 \to t_2|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool, t_2 \to t_2 = t_3 \to int|\!)_\mathcal{E}$
- $(\!|t_2 \to t_2|\!)_k (\!|f = let(t \to t)|\!)_\Gamma (\!|t_1 = bool, t_2 = t_3, t_2 = int|\!)_\mathcal{E}$
- $int \to int$

Constraints solvable: $t_1 = bool$, $t_2 = t_3 = int$.
Final type: $int \to int$.

Unification à la Martelli & Montanari (1982)

- Divide & Conquer: Decompose algebraic structure to basic constraints & substitute them inside other constraints.

$$(t \equiv t) \rightarrow \cdot \quad (1)$$

$$(t_1 \mapsto t_2 \equiv t'_1 \mapsto t'_2) \rightarrow (t_1 \equiv t'_1),\ (t_2 \equiv t'_2) \quad (2)$$

$$(t \equiv t_v) \rightarrow (t_v \equiv t) \quad \text{when } t \notin \mathit{TypeVar} \quad (3)$$

$$t_v \equiv t,\ t_v \equiv t' \rightarrow t_v \equiv t,\ t \equiv t' \quad \text{when } t, t' \neq t_v \quad (4)$$

$$t_v \equiv t,\ t'_v \equiv t' \rightarrow t_v \equiv t,\ t'_v \equiv t'[t_v \leftarrow t]$$
$$\text{when } t_v \neq t'_v,\ t_v \neq t,\ t'_v \neq t',\ \text{and } t_v \in \mathit{vars}(t') \quad (5)$$

- Set of unifiers is an invariant for each rule
- Rules are ground confluent and decreasing, computing MGU.

$\mathcal{W}$ Definition

- Put the program to be typed in the initial environment

$$[\![e]\!] = \langle\!| \langle\!| e |\!\rangle_k \ \langle\!| \cdot |\!\rangle_{tenv} \ \langle\!| \cdot |\!\rangle_{eqns} \ \langle\!| t_0 |\!\rangle_{nextType} |\!\rangle_\top \tag{6}$$

- Once the program "evaluated" to a type, resolve it using accumulated constraints

$$\langle \langle\!| t |\!\rangle_k \ \langle\!| \gamma |\!\rangle_{eqns} \rangle_\top = \gamma[t] \tag{7}$$

$\mathcal{W}$ Definition (continued)

- Constants evaluate to their types

$$\begin{gathered} i \rightarrow int,\ true \rightarrow bool,\ false \rightarrow bool \\ \text{(and similarly for all the other basic values)} \end{gathered} \tag{8}$$

- Sum evaluates to int; adds constraint that its parameters are ints

$$\left(\!\left|\frac{t_1 + t_2}{int}\right|\!\right)_k \left\langle \frac{\cdot}{t_1 \equiv int,\ t_2 \equiv int} \right\rangle_{eqns} \tag{9}$$

$\mathcal{W}$ Definition (continued)

- A fresh type variable is bound to x, and e is type in that environment. Environment must be restored afterwards.

$$\left(\!\left|\frac{\lambda x.e}{(t_v \rightarrow e) \curvearrowright restore(\eta)}\right|\!\right)_k \left(\!\left|\frac{\eta}{\eta[x \leftarrow t_v]}\right|\!\right)_{tenv} \left(\!\left|\frac{t_v}{t_v + 1}\right|\!\right)_{nextType} \quad (10)$$

- Application: t_1 is constraint to be the function type taking t_2 as input and producing t_v, a new type variable.

$$\left(\!\left|\frac{t_1\ t_2}{t_v}\right|\!\right)_k \left\langle\!\left|\frac{\cdot}{t_1 \equiv t_2 \rightarrow t_v}\right|\!\right\rangle_{eqns} \left(\!\left|\frac{t_v}{t_v + 1}\right|\!\right)_{nextType} \quad (11)$$

$\mathcal{W}$ Definition (continued)

- If statement: constrain branches to have same type, condition to bool

$$\left(\!\!\left|\frac{\text{if } t \text{ then } t_1 \text{ else } t_2}{t_1}\right|\!\!\right)_k \left\langle\!\left|\frac{\cdot}{t \equiv bool,\ t_1 \equiv t_2}\right|\!\right\rangle_{eqns} \tag{12}$$

- Let: bind x to the special type t and evaluate e. Restore environment after.

$$\left(\!\!\left|\frac{\text{let } x = t \text{ in } e}{e \curvearrowright restore(\eta)}\right|\!\!\right)_k \left(\!\!\left|\frac{\eta}{\eta[x \leftarrow let(t)]}\right|\!\!\right)_{env} \tag{13}$$

$\mathcal{W}$ Definition (continued)

- If variable is bound to simple type, just instantiate it

$$\left(\!\left| \frac{x}{\eta[x]} \right|\!\right)_k \left(\!\left| \eta \right|\!\right)_{tenv} \quad \text{when } \eta[x] \neq let(t) \tag{14}$$

- If variable is bound to a let type, first resolve the type, then replace free variables by fresh copies and use that type for the variable

$$\left(\!\left| \frac{x}{(\gamma[t])[tl \leftarrow tl']} \right|\!\right)_k \left(\!\left| \eta \right|\!\right)_{tenv} \left(\!\left| \gamma \right|\!\right)_{eqns} \left(\!\left| \frac{t_v}{t_v + |tl|} \right|\!\right)_{nextType}$$
$$\text{when } \eta[x] = let(t),\ tl = vars(\gamma[t]) - vars(\eta)$$
$$\text{and } tl' = t_v \ldots (t_v + |tl| - 1) \tag{15}$$

How about the execution time?

- K definition of $\mathcal{W}$ simpler than Milner's original $\mathcal{W}$ algorithm
- Comparable (in speed) with existing inference algorithms
- Stress test program (polymorphic in $2^n + 1$ type variables!):

$$
\begin{array}{l}
\text{let } f_0 = \lambda x.\lambda y.x \text{ in} \\
\quad \text{let } f_1 = \lambda x.f_0(f_0 x) \text{ in} \\
\quad\quad \text{let } f_2 = \lambda x.f_1(f_1 x) \text{ in} \\
\quad\quad\quad \ldots \\
\quad\quad\quad\quad \text{let } f_n = \lambda x.f_{n-1}(f_{n-1} x) \text{ in } f_n
\end{array}
$$

Speed of various $\mathcal{W}$ implementations

-	n = 10		n = 12		n = 14	
OCAML (version 3.09.3)	0.6s	3M	8.3s	5M	124.9s	13M
Haskell (ghci version 6.8.1)	1.5s	25M	21.8s	31M	614.7s	61M
SML (version 110.59)	4.9s	76M	111.4s	324M	internal error	
$\mathcal{W}$ in K/Maude2.3 with memo	1.4s	11M	23.8s	70M	395.9s	653M
$\mathcal{W}$ in K/Maude2.3 without memo	2.5s	10M	26.2s	51M	367.5s	574M
!$\mathcal{W}$ in K/Maude2.3 with memo	1.4s	12M	22.8s	70M	377.4s	654M
!$\mathcal{W}$ in K/Maude2.3 without memo	2.4s	11M	26.0s	52M	359.6s	575M
$\mathcal{W}$ in PLT/Redex	$>$1h		-		-	
$\mathcal{W}$ in OCAML	105.9s	1.9M	$>$1h	2.7M	-	

- Ratios appear to scale and are preserved for other programs
- No slowdown !$\mathcal{W}$ is an extension of $\mathcal{W}$ with lists, products, side effects (through refs and assignment) and weak polymorphism.
- Memoization pays off when polymorphic types are small

Outline

Proof Technique Motivation

- K uses rewriting logic, as opposed to reduction semantics:
  - Preservation traditionally requires context-rewriting.
  - Intermediate configurations are a mish-mash of language syntax and types.
  - Handled by relating language configurations with type configurations.
- Additionally, K-style definitions are concrete:
  - Written to provide an interpreter immediately.
  - Properties that are true modulo concrete details are complex.
  - Handled by constructing an abstract type system.

Description of Morphism α

- We build a function that correlates partially evaluated programs with their types.
- Generalization of the syntax driven approach.
  - $\alpha([\![E]\!]_{\mathcal{L}}) = [\![E]\!]_{\mathcal{T}}$
  - The above works when there is exactly one expression per configuration equivalence class.
- Technique used frequently in the domain of processor construction and compiler optimization.

Abstract Type Inferencer

- Wanted to work directly on the type system definition itself, while also working modulo:
  - Alpha equivalence: handled by a bijection
  - Equivalent unifiers: handled by a canonical unifier
  - Unifiable configuration fragments: handled by composing unification with each rewrite

Statement of Preservation

- Preservation: If $[\![E]\!]_{\mathcal{T}} \xrightarrow{*} \tau$ and $[\![E]\!]_{\mathcal{L}} \xrightarrow{*} V$ for some type τ and value V, then $[\![V]\!]_{\mathcal{T}} \xrightarrow{*}$ some τ'.
  1. Main Lemma: If $[\![E]\!]_{\mathcal{T}} \xrightarrow{*} \tau$ and $[\![E]\!]_{\mathcal{L}} \xrightarrow{*} R$ for some τ and R, then $\mathcal{T} \models \alpha(R) \xrightarrow{*} \tau'$ for some τ'.
  2. Secondary Lemma: If $\mathcal{T} \models \alpha(V) \xrightarrow{*} \tau$ then $[\![V]\!]_{\mathcal{T}} \xrightarrow{*} \tau$.
- In comparison, the definition of preservation as given by Wright and Felleisen states: "If $\Gamma \triangleright e_1 : \tau$ and $e_1 \longrightarrow e_2$ then $\Gamma \triangleright e_2$: some τ'."

Outline

Conclusion

- Formal, executable definition of Milner's Exp & $\mathcal{W}$:
  - Mathematically precise description of language and inferencer.
  - Uses the same formalism for both.
  - Inferencer execution time comparable to real implementations.
- Type preservation proof techniques:
  - Morphism from language configurations to typing configurations
  - Abstract type system

Future Work

- Formalize the proof of type preservation in a proof assistant.
- Develop a library of lemmas useful across different proofs.

Thank you!

Backup Slides

Main Lemma for Preservation

If $[\![E]\!]_{\mathcal{T}} \xrightarrow{*} \tau$ and $[\![E]\!]_{\mathcal{L}} \xrightarrow{*} R$ for some τ and R, then $\mathcal{T} \models \alpha(R) \xrightarrow{*} \tau'$ for some τ'.
The proof proceeds by induction over the steps taken to get from $[\![E]\!]_{\mathcal{L}}$ to R.

Base Case Assume no steps were taken. Then $R = [\![E]\!]_{\mathcal{L}}$. By the definition of α, we see that $\alpha(R) = [\![E]\!]_{\mathcal{T}}$. By assumption, this reduces to τ, so we have that $\alpha(R) \xrightarrow{*} \tau$.

Induction Case Assume $[\![E]\!]_{\mathcal{L}} \xrightarrow{n} R$ and $\alpha(R) \xrightarrow{*} \tau$. Assume an $n + 1$ step can be taken to get to a state R'. This step could be any one of the structural or semantic rules of the language. We consider each individually. Below we give an example of one of the cases.

Example Case

$R = (\!|(\!|I : Int + I' : Int \curvearrowright K|\!)_{k_{\mathcal{L}}}|\!)_{\top}$ First we work with $\alpha(R)$:

$$\alpha(R) = \alpha((\!|(\!|I + I' \curvearrowright K|\!)_{k_{\mathcal{L}}}|\!)_{\top})$$

which reduces to:

$$(\!|(\!|I + I' \curvearrowright K|\!)_{k_{\mathcal{T}}} \ (\!|\cdot|\!)_{env_{\mathcal{T}}} \ (\!|\cdot|\!)_{eqns} \ (\!|\tau_0|\!)_{nextType}|\!)_{\top}$$

by the definition of α. This then reduces to:

$$(\!|(\!|\textsc{Int} + \textsc{Int} \curvearrowright K|\!)_{k_{\mathcal{T}}} \ (\!|\cdot|\!)_{env_{\mathcal{T}}} \ (\!|\cdot|\!)_{eqns} \ (\!|\tau_0|\!)_{nextType}|\!)_{\top}$$

because we reduce integers to Int. This then reduces to:

$$(\!|(\!|\textsc{Int} \curvearrowright K|\!)_{k_{\mathcal{T}}} \ (\!|\cdot|\!)_{env_{\mathcal{T}}} \ (\!|\textsc{Int} = \textsc{Int}, \textsc{Int} = \textsc{Int}|\!)_{eqns} \ (\!|\tau_0|\!)_{nextType}|\!)_{\top}$$

by applying the reduction rule for addition. Finally, we can reduce this to:

$$(\!|(\!|\textsc{Int} \curvearrowright K|\!)_{k_{\mathcal{T}}} \ (\!|\cdot|\!)_{env_{\mathcal{T}}} \ (\!|\cdot|\!)_{eqns} \ (\!|\tau_0|\!)_{nextType}|\!)_{\top}$$

by applying one of the rules of unification twice.

Example Case Cont.

Now we work with R'. We start with:

$$\alpha(R') = \alpha(\llparenthesis \llparenthesis I +_{int} I' \curvearrowright K \rrparenthesis_{k_{\mathcal{L}}} \rrparenthesis_{\top})$$

which reduces to:

$$\llparenthesis \llparenthesis I +_{int} I' \curvearrowright K \rrparenthesis_{k_{\mathcal{T}}} \llparenthesis \cdot \rrparenthesis_{env_{\mathcal{T}}} \llparenthesis \cdot \rrparenthesis_{eqns} \llparenthesis \tau_0 \rrparenthesis_{nextType} \rrparenthesis_{\top}$$

by the definition of α. This immediately reduces to:

$$\llparenthesis \llparenthesis \text{INT} \curvearrowright K \rrparenthesis_{k_{\mathcal{T}}} \llparenthesis \cdot \rrparenthesis_{env_{\mathcal{T}}} \llparenthesis \cdot \rrparenthesis_{eqns} \llparenthesis \tau_0 \rrparenthesis_{nextType} \rrparenthesis_{\top}$$

because we reduce integers to INT. So, we now have that $\alpha(R)$ and $\alpha(R')$ both reduce to the same configuration. We know by inductive assumption that $\alpha(R) \xrightarrow{*} \tau$. Since $\alpha(R)$ and $\alpha(R')$ both reduce to the same configuration, $\alpha(R) \xrightarrow{*} \tau$ also. This completes the case.