A Rewriting Logic Approach to Type Inference

Chucky Ellison, Traian Florin Şerbănuţă and Grigore Roşu

Department of Computer Science, University of Illinois at Urbana-Champaign
{celliso2, tserban2, grosu}@cs.uiuc.edu

Rewriting logic semantics (RLS) was proposed as a programing language definitional framework that unifies operational and algebraic denotational semantics; see [5, 7] and the references there. Once a language is defined as an RLS theory, many generic tools are immediately available for use with no additional cost to the designer. These include a formal inductive theorem proving environment, an efficient interpreter, a state space explorer, and even a model checker. RLS has already been used to define a series of didactic and real languages [5].

In its SOS'05 precursor, [5] proposed using the same rewriting logic technique to define type systems and policy checkers for languages; more precisely, to rewrite integer values to their types and to maintain and incrementally rewrite a program until it becomes a type or other desired abstract value. That idea was further explored by in [7], but not yet used to define complex, polymorphic type systems; it also provides no implementation, no proofs, and no empirical evaluation of the idea. A similar idea has been recently proposed by [3] in the context of Felleisen et al.'s reduction semantics with evaluation contexts [2, 8] and Matthews et al.'s PLT Redex system [4].

In this paper we show how the same rewriting logic semantics (RLS) framework and definitional style employed in giving formal semantics to languages can also be used to define type systems as rewrite logic theories. This way, both the language and its type system(s) can be defined using the same formalism, facilitating reasoning about programs, languages, and type systems.

We use Milner's polymorphic type inferencer $\mathcal{W}$ [6] for the Exp language to exemplify our technique. We give one rewrite logic theory for $\mathcal{W}$ and use it both for proving its correctness against a rewrite theory defining Exp and for obtaining an efficient, executable type-inferencer.

Our definitional style gains modularity by specifying the minimum amount of information needed for a transition to occur, and compositionality by using *strictness* attributes associated to the language constructs, which specify that the semantics of that construct involves certain specified arguments to be evaluated (and their side effects propagated) prior to giving semantics to the construct itself. These allow us, for example, to have the rule for function application corresponding to the one in $\mathcal{W}$ look as follows (assuming the application was declared strict in both arguments):

$$\langle\!\langle \frac{t_1\ t_2}{tvar} \rangle\!\rangle_k \ \langle\!\langle \frac{\cdot}{t_1 \equiv t_2 \to tvar} \rangle\!\rangle_{eqns} \quad \text{where } tvar \text{ is a fresh type variable}$$

which reads as follows: once all constraints for both sides of an application construct were gathered, the application of t_1 to t_2 will have a new type, *tvar*, with the additional constraint that t_1 is the function type $t_2 \to tvar$.

This work makes three novel contributions:

1. It shows how non-trivial type systems are defined as RLS theories in a uniform way, following the same style used for defining programming languages and other formal analyses for them;
2. It proposes a type soundness proof technique for languages and type systems defined as RLS theories; and
3. It shows that RLS definitions of type systems, when executed on existing rewrite engines, yield competitive type inferencers.

To show that the proposed rewriting approach and the resulting type inferencers scale, Milner's simple language is extended with multiple-binding let and letrec, with lists, and with references and side effects. The resulting type inferencer, able to detect weak polymorphism, is only slightly slower than the one for Milner's simpler language.

All these show that rewriting logic is amenable for defining feasible type inferencers for programming languages and proving type soundness for those definitions. Doing the proof of soundness for $\mathcal{W}$ and other systems have led us to believe that this kind of proofs should be easily mechanizable. We strongly adhere to the program proposed by the PoplMark Challenge [1], and would like to approach it using the proposed novel methodology.

Below we include the K definition of the $\mathcal{W}$ type inferencer, including the syntax of Exp for $\mathcal{W}$, the configuration, unification rules, and complete typing rules. Information about the K syntax can be found in [7].

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

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

$$
\begin{array}{ll}
(t \equiv t) \to \cdot & \\
(t_1 \mapsto t_2 \equiv t_1' \mapsto t_2') \to (t_1 \equiv t_1'),\ (t_2 \equiv t_2') & \\
(t \equiv t_v) \to (t_v \equiv t) & \text{when } t \notin \mathit{TypeVar} \\
t_v \equiv t,\ t_v \equiv t' \to t_v \equiv t,\ t \equiv t' & \text{when } t, t' \neq t_v
\end{array}
$$

$t_v \equiv t,\ t'_v \equiv t' \rightarrow t_v \equiv t,\ t'_v \equiv t'[t_v \leftarrow t]$
when $t_v \neq t'_v,\ t_v \neq t,\ t'_v \neq t',$ and $t_v \in vars(t')$

$[\![e]\!] = [\![(\!|e|\!)_k\ (\!|\cdot|\!)_{tenv}\ (\!|\cdot|\!)_{eqns}\ (\!|t_0|\!)_{nextType}]\!]$
$[\![\langle\!|(\!|t|\!)_k\ (\!|\gamma|\!)_{eqns}|\!\rangle]\!] = \gamma[t]$
$i \rightarrow int,\ true \rightarrow bool,\ false \rightarrow bool,$ (and similarly for all the other basic values)
$(\!|\underfrac{t_1 + t_2}{int}|\!)_k\ \langle\!|\underfrac{\cdot}{t_1 \equiv int,\ t_2 \equiv int}|\!\rangle_{eqns}$ (and similarly for all the other standard operators)
$(\!|\underfrac{x}{(\gamma[t])[tl \leftarrow tl']}|\!)_k\ (\!|\eta|\!)_{tenv}\ (\!|\gamma|\!)_{eqns}\ (\!|\underfrac{t_v}{t_v + |tl|}|\!)_{nextType}$
when $\eta[x] = let(t),\ tl = vars(\gamma[t]) - vars(\eta)$
and $tl' = t_v \ldots (t_v + |tl| - 1)$ $(\!|\underfrac{x}{\eta[x]}|\!)_k\ (\!|\eta|\!)_{tenv}$ when $\eta[x] \neq let(t)$
$(\!|\underfrac{\lambda x.e}{(t_v \rightarrow e) \curvearrowright restore(\eta)}|\!)_k\ (\!|\underfrac{\eta}{\eta[x \leftarrow t_v]}|\!)_{tenv}\ (\!|\underfrac{t_v}{t_v + 1}|\!)_{nextType}$
$(\!|\underfrac{t_1\ t_2}{t_v}|\!)_k\ \langle\!|\underfrac{\cdot}{t_1 \equiv t_2 \rightarrow t_v}|\!\rangle_{eqns}\ (\!|\underfrac{t_v}{t_v + 1}|\!)_{nextType}$
$(\!|\underfrac{\mu x.e}{e \curvearrowright ?_=(t_v) \curvearrowright restore(\eta)}|\!)_k\ (\!|\underfrac{\eta}{\eta[x \leftarrow t_v]}|\!)_{tenv}\ (\!|\underfrac{t_v}{t_v + 1}|\!)_{nextType}$
$(\!|t \rightarrow \underfrac{?_=t_v}{\cdot}|\!)_k\ \langle\!|\underfrac{\cdot}{t_v \equiv t}|\!\rangle_{eqns}$
$(\!|\underfrac{\mathsf{let}\ x = t\ \mathsf{in}\ e}{e \curvearrowright restore(\eta)}|\!)_k\ (\!|\underfrac{\eta}{\eta[x \leftarrow let(t)]}|\!)_{env}$
$(\!|\underfrac{\mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2}{t_1}|\!)_k\ \langle\!|\underfrac{\cdot}{t \equiv bool,\ t_1 \equiv t_2}|\!\rangle_{eqns}$

References

1. B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In *TPHOLs '05*, volume 3603 of *LNCS*, pages 50–65. Springer, 2005.
2. M. Felleisen and R. Hieb. A revised report on the syntactic theories of sequential control and state. *J. TCS*, 103(2):235–271, 1992.
3. G. Kuan, D. MacQueen, and R. B. Findler. A rewriting semantics for type inference. In *ESOP '07*, volume 4421 of *LNCS*, pages 426–440. Springer, 2007.
4. J. Matthews, R. B. Findler, M. Flatt, and M. Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *RTA '04*, volume 3091 of *LNCS*, pages 301–311. Springer, 2004.
5. J. Meseguer and G. Rosu. The rewriting logic semantics project. *J. TCS*, 373(3):213–237, 2007.
6. R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17(3):348–375, 1978.
7. G. Rosu. K: A rewrite-based framework for modular language design, semantics, analysis and implementation. Technical Report UIUCDCS-R-2006-2802, Computer Science Department, University of Illinois at Urbana-Champaign, 2006.
8. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.