WADT'08, LNCS 5486, pp 135-151. 2009

A Rewriting Logic Approach to Type Inference[*]

Chucky Ellison, Traian Florin Șerbănuță and Grigore Roșu

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

Abstract. Meseguer and Roșu proposed rewriting logic semantics (RLS) as a programing language definitional framework that unifies operational and algebraic denotational semantics. RLS has already been used to define a series of didactic and real languages, but its benefits in connection with defining and reasoning about type systems have not been fully investigated. This paper shows how the same RLS style employed for giving formal definitions of languages can be used to define type systems. The same term-rewriting mechanism used to execute RLS language definitions can now be used to execute type systems, giving type checkers or type inferencers. The proposed approach is exemplified by defining the Hindley-Milner polymorphic type inferencer $\mathscr{W}$ as a rewrite logic theory and using this definition to obtain a type inferencer by executing it in a rewriting logic engine. The inferencer obtained this way compares favorably with other definitions or implementations of $\mathscr{W}$. The performance of the executable definition is within an order of magnitude of that of highly optimized implementations of type inferencers, such as that of OCaml.

1 Introduction

Meseguer and Roșu proposed rewriting logic as a semantic foundation for the definition and analysis of languages [1, 2], as well as type systems and policy checkers for languages [2]. More precisely, they proposed rewriting integer values to their types and incrementally rewriting a program until it becomes a type or other desired abstract value. That idea was further explored by Roșu [3], but not used to define polymorphic type systems. Also, no implementation, no proofs, and no empirical evaluation of the idea were provided. A similar idea has been recently proposed by Kuan, MacQueen, and Findler [4] in the context of Felleisen et al.'s reduction semantics with evaluation contexts [5, 6] and Matthews et al.'s PLT Redex system [7].

In this paper we show how the same rewriting logic semantics (RLS) framework and definitional style employed in giving formal semantics to languages can be used to also 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.

[*] Supported in part by NSF grants CCF-0448501, CNS-0509321 and CNS-0720512, by NASA contract NNL08AA23C, by the Microsoft/Intel funded Universal Parallel Computing Research Center at UIUC, and by several Microsoft gifts.

We use the Hindley-Milner polymorphic type inferencer $\mathscr{W}$ [8] for Milner's Exp language to exemplify our technique. We give one rewrite logic theory for $\mathscr{W}$ and use it to obtain 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 with the language constructs. These allow us, for example, to have the rule for function application corresponding to the one in $\mathscr{W}$ look as follows (assuming application was declared strict in both arguments):

$$(\!|\underset{tvar}{\underline{t_1\ t_2}}\rangle_k\ \langle\!|\underset{t_1 = t_2 \to tvar}{\underline{\qquad\cdot\qquad}}|\!\rangle_{eqns} \quad \text{where } tvar \text{ is a fresh type variable}$$

which reads as follows: once all constraints for both sides of an application construct are 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 paper makes two novel contributions:

1. It shows how non-trivial type systems are defined as RLS theories, following the same style used for defining languages and other formal analyses;
2. It shows that RLS definitions of type systems, when executed on existing rewrite engines, yield competitive type inferencers.

Related work. In addition to the work mentioned above, there has been other previous work combining term rewriting with type systems. For example, the Stratego reference manual [9] describes a method of using rewriting to add type-check notations to a program. Also, pure type systems, which are a generalization of the λ-cube [10], have been represented in membership equational logic [11], a subset of rewriting logic. There is a large body on term graph rewriting [12, 13] and its applications to type systems [14, 15]. There are similarities with our work, such as using a similar syntax for both types and terms, and a process of reduction or normalization to reduce programs to their types. A collection of theoretical papers on type theory and term rewriting can be found in [16]. Adding rewrite rules as annotations to a particular language in order to assist a separate algorithm with type checking has been explored [17], as well as adding type annotations to rewrite rules that define program transformations [18]. Much work has been done on defining type systems modularly [19–23]. The style we propose in this paper is different from previous approaches combining term rewriting with type systems. Specifically, we use an executable definitional style within rewriting logic semantics, called K [3, 24]. The use of K makes the defined type inferencers easy to read and understand, as well as efficient when executed.

Section 2 introduces RLS and the K definitional style, and gives an RLS definition of Milner's Exp language. Section 3 defines the Hindley-Milner $\mathscr{W}$ algorithm as an RLS theory and reports on some experiments. Section 4 shows that our RLS definition faithfully captures the Hindley-Milner algorithm, and gives a summary of preservation results. Section 5 concludes the paper.

WADT'08, LNCS 5486, pp 135-151. 2009

2 Rewriting Semantics

This section recalls the RLS project, then presents the K technique for designing programming languages. We briefly discuss these, and show how Milner's Exp language [8] can be defined as an RLS theory using K. The rest of the paper employs the same technique to define type systems as RLS theories.

2.1 Rewriting Logic

Term rewriting is a standard computational model supported by many systems. Meseguer's rewriting logic [25], not to be confused with term rewriting, organizes term rewriting modulo equations as a logic with a complete proof system and initial model semantics. Meseguer and Roșu's RLS [1, 2] seeks to make rewriting logic a foundation for programming language semantics and analysis that unifies operational and algebraic denotational semantics.

In contrast to term rewriting, which is just a method of computation, rewriting logic is a computational logic built upon equational logic, proposed by Meseguer [25] as a logic for true concurrency. In equational logic, a number of *sorts* (types) and *equations* are defined, specifying which terms are to be considered equivalent. Rewriting logic adds *rules* to equational logic, thought of as irreversible transitions: a rewrite theory is an equational theory extended with rewrite rules. Rewriting logic admits a complete proof system and an initial model semantics [25] that makes inductive proofs valid. Rewriting logic is connected to term rewriting in that all the equations $l = r$ can be transformed into term rewriting rules $l \rightarrow r$. This provides a means of taking a rewriting logic theory, together with an initial term, and "executing" it. Any of the existing rewrite engines can be used for this purpose. Some of the engines, e.g., Maude [26], provide even richer support than execution, such as an inductive theorem prover, a state space exploration tool, a model checker, and more.

RLS builds upon the observation that programming languages can be defined as rewrite logic theories. By doing so, one gets "for free" not only an interpreter and an initial model semantics for the defined language, but also a series of formal analysis tools obtained as instances of existing tools for rewriting logic. Operationally speaking, the major difference between conventional reduction semantics, with [5] or without [27] evaluation contexts, and RLS is that the former typically impose contextual restrictions on applications of reduction steps and the reduction steps happen one at a time, while the latter imposes no such restrictions. To avoid undesired applications of rewrite steps, one has to obey certain methodologies when using rewriting logic. In particular, one can capture the conventional definitional styles by appropriate uses of conditional rules. Consequently, one can define a language many different ways in rewriting logic. In this paper, we use Roșu's K technique [3], which is inspired by abstract state machines [28] and continuations [29], and which glosses over many rewriting logic details that are irrelevant for programming languages. Roșu's K language definitional style optimizes the use of RLS by means of a definitional technique and a specialized notation.

2.2 K

The idea underlying K is to represent the program configuration as a nested "soup" (multiset) of *configuration item* terms, also called *configuration cells*, representing the current infrastructure needed to process the remaining program or fragment of program; these may include the current computation (a continuation-like structure), environment, store, remaining input, output, analysis results, bookkeeping information, etc. The set of configuration cells is not fixed and is typically different from definition to definition. K assumes lists, sets and multisets over any sort whenever needed; for a sort S, $\mathsf{List}[S]$ denotes comma-separated lists of terms of sort S, and $\mathsf{Set}[S]$ denotes white space separated sets of terms of sort S. For both lists and sets, we use "$\cdot$" as unit (nil, empty, etc.). To use a particular list- or set-separator, one writes it as an index; for example, $\mathsf{List}_{\curvearrowright}[S]$ stands for $\curvearrowright$-separated lists of terms of sort S. Lists and sets admit straightforward equational definitions in rewriting logic (a list is an associative binary operator, while a set is an associative, commutative, and idempotent binary operator). Formally, configurations have the following structure:

$$
\begin{aligned}
\mathit{ConfigLabel} &::= \top \mid k \mid \mathit{env} \mid \mathit{store} \mid ... \\
&\quad\text{(descriptive names; first two common, rest differ with language)} \\
\mathit{Config} &::= [\![K]\!] \mid ... \mid (\!|S|\!)_{\mathit{ConfigLabel}} \\
&\quad\text{(S can be any sort, including $\mathsf{Set}[\mathit{Config}]$)}
\end{aligned}
$$

The advantage of representing configurations as nested "soups" is that language rules only need to mention applicable configuration cells. This is one aspect of K's modularity. We can add or remove elements from the configuration set as we like, only impacting rules that use those particular items. Rules do not need to be changed to match what the new configuration looks like.

Almost all definitions share the configuration labels $\top$ (which stands for "*top*") and k (which stands for "*current computation*"). The remaining configuration labels typically differ with the language or analysis technique to be defined in K. A configuration $(\!|c|\!)_l$ may also be called a *configuration item* (or *cell*) *named* (or *labeled*) l; interesting configuration cells are the nested ones, namely those where $c \in \mathsf{Set}[\mathit{Config}]$. One is also allowed to define some language-specific configuration constructs, to more elegantly intialize and terminate computations. A common such additional configuration construct is $[\![p]\!]$, which takes the given program to an initial configuration. An equation therefore needs to be given, taking such special initializing configuration into an actual configuration cell; in most definitions, an equation identifies a term of the form $[\![p]\!]$ with one of the form $(\!|(\!|p|\!)_k ...|\!)_\top$, for some appropriate configuration items replacing the dots. If one's goal is to give a dynamic semantics of a language and if p is some terminating program, then $[\![p]\!]$ eventually produces (after a series of rewrites) the result of evaluating p; if one's goal is to analyze p, for example to type check it, then $[\![p]\!]$ eventually rewrites to the result of the analysis, for example a type when p is well-typed or an error term when p is not well-typed.

The most important configuration item, present in all K definitions and "wrapped" with the *ConfigLabel* k, is the *computation*, denoted by K. Computations generalize abstract syntax by adding a special list construct (associative operator) $\_ \curvearrowright \_$:

$$\begin{aligned}
K &::= \mathit{KResult} \mid \mathit{KLabel}(\mathsf{List}[K]) \mid \mathsf{List}_{\curvearrowright}[K] \\
\mathit{KResult} &::= \text{(finished computations, e.g., values, or types, etc.)} \\
\mathit{KLabel} &::= \text{(one per language construct)}
\end{aligned}$$

The construct $\mathit{KLabel}(\mathsf{List}[K])$ captures any programming language syntax, under the form of an abstract syntax tree. If one wants more K syntactic categories, then one can do that, too, but we prefer to keep only one here. In our Maude implementation, thanks to Maude's mixfix notation for syntax, we write, e.g., "$\mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$" in our definitions instead of "$\mathsf{if\_then\_else\_}(b, s_1, s_2)$".

The distinctive K feature is $\_ \curvearrowright \_$. Intuitively, $k_1 \curvearrowright k_2$ says "process k_1 then k_2". How this is used and what the meaning of "process" is left open and depends upon the particular definition. For example, in a concrete semantic language definition it can mean "evaluate k_1 then k_2", while in a type inferencer definition it can mean "type and accumulate type constraints in k_1 then in k_2". A K definition consists of two types of sentences: structural equations and rewrite rules. Structural equations carry no computational meaning; they only say which terms should be viewed as identical and their role is to transparently modify the term so that rewrite rules can apply. Rewrite rules are seen as irreversible computational steps and can happen concurrently on a match-and-apply basis. The following are examples of structural equations:

$$\begin{aligned}
&a_1 + a_2 = a_1 \curvearrowright \square + a_2 \\
&a_1 + a_2 = a_2 \curvearrowright a_1 + \square \\
&\mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 = b \curvearrowright \mathsf{if}\ \square\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2
\end{aligned}$$

Note that, unlike in evaluation contexts, $\square$ is not a "hole," but rather part of a *KLabel*, carrying the obvious "plug" intuition; e.g., the *KLabel*s involving $\square$ above are $\square + \_$, $\_ + \square$, and $\mathsf{if}\ \square\ \mathsf{then\_else\_}$, respectively. To avoid writing such obvious, distracting, and mechanical structural equations, the language syntax can be annotated with *strict* attributes when defining language constructs: a *strict* construct is associated with an equation as above for each of its subexpressions. If an operator is intended to be strict in only some of its arguments, then the positions of the strict arguments are listed as arguments of the *strict* attribute; for example, the above three equations correspond to the attributes *strict* for $\_ + \_$ and *strict*(1) for $\mathsf{if\_then\_else\_}$. All these structural equations are automatically generated from strictness attributes in our implementation.

Structural equations can be applied back and forth; for example, the first equation for $\_ + \_$ can be applied left-to-right to "schedule" a_1 for processing; once evaluated to i_1, the equation is applied in reverse to "plug" the result back in context, then a_2 is scheduled with the second equation left-to-right, then its result i_2 plugged back into context, and then finally the rewrite rule can apply the irreversible computational step. Special care must be taken so that side effects are propagated appropriately: they are only generated at the leftmost side of the computation.

The following are examples of rewrite rules:

$$\begin{array}{l}
i_1 + i_2 \rightarrow i, \text{ where } i \text{ is the sum of } i_1 \text{ and } i_2 \\
\textsf{if true then } s_1 \textsf{ else } s_2 \rightarrow s_1 \\
\textsf{if false then } s_1 \textsf{ else } s_2 \rightarrow s_2
\end{array}$$

In contrast to structural equations, rewrite rules can only be applied left to right.

2.3 A Concrete Example: Milner's Exp Language

Milner proved the correctness of his $\mathscr{W}$ type inferencer in the context of a simple higher-order language that he called Exp [8]. Recall that $\mathscr{W}$ is the basis for the type checkers of all statically typed functional languages.

Exp is a simple expression language containing lambda abstraction and application, conditional, fix point, and "let" and "letrec" binders. To exemplify K and also to remind the reader of Milner's Exp language, we next define it using K. Figure 1 shows its K annotated syntax and Figure 2 shows its K

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

Fig. 1. K-Annotated Syntax of Exp

$$\begin{array}{rcl}
\mathit{Val} & ::= & \lambda\, \mathit{Var}.\, \mathit{Exp} \mid \text{...(Bools, ints, etc.)} \\
\mathit{KResult} & ::= & \mathit{Val} \\
\mathit{Config} & ::= & \mathit{Val} \mid [\![K]\!] \mid \llparenthesis K \rrparenthesis_k
\end{array}
\qquad
\begin{array}{l}
[\![e]\!] = \llparenthesis e \rrparenthesis_k \\
\llparenthesis v \rrparenthesis_k = v \\
\llparenthesis \dfrac{(\lambda x.e)\, v}{e[x \leftarrow v]} \rrparenthesis_k \\
\llparenthesis \dfrac{\mu\ x.e}{e[x \leftarrow \mu\ x.e]} \rrparenthesis_k \\
\textsf{if true then } e_1 \textsf{ else } e_2 \rightarrow e_1 \\
\textsf{if false then } e_1 \textsf{ else } e_2 \rightarrow e_2
\end{array}$$

Fig. 2. K Configuration and Semantics of Exp

semantics. We also use this to point out some other K notational conventions. Note that application is strict in both its arguments (call-by-value) and that let and letrec are desugared. Additionally, syntactic constructs may be annotated with desugaring equations. In Figure 2, we see that λ-abstractions are defined as values, which are also *KResult*s in this definition; *KResult*s are not further "scheduled" for processing in structural equations. Since Exp is simple, there is only one *ConfigItem* needed, wrapped by *ConfigLabel* k. The first two equations initialize and terminate the computation process. The third applies the β-reduction when $(\lambda x.e)\,v$ is the first item in the computation; we here use two other pieces of K notation: list/set fragment matching and the two-dimensional writing for rules. The first allows us to use angle brackets for unimportant fragments of lists or sets; for example, $(\!|T\rangle$ matches a list whose prefix is T, $\langle T|\!)$ matches a list whose suffix is T, and $\langle T\rangle$ matches a list containing a contiguous fragment T; same for sets, but the three have the same meaning there. Therefore, special parentheses $(\!|$ and $|\!)$ represent respective ends of a list/set, while angled variants mean "the rest." The second allows us to avoid repetition of contexts; for example, instead of writing a rule of the form $C[t_1, t_2, ..., t_n] \rightarrow C[t'_1, t'_2, ..., t'_n]$ (rewriting the above-mentioned subterms in context C) listing the context (which can be quite large) C twice, we can write it $C[\underset{t'_1}{\underline{t_1}}, \underset{t'_2}{\underline{t_2}}, ..., \underset{t'_n}{\underline{t_n}}]$ with the obvious meaning, mentioning the context only once. The remaining Exp semantics is straightforward. Note that we used the conventional substitution, which is also provided in our Maude implementation.

The Exp syntax and semantics defined in Figures 1 and 2 is all we need to write in our implementation of K. To test the semantics, one can now execute programs against the obtained interpreter.

3 Defining Milner's $\mathscr{W}$ Type Inferencer

We next define Milner's $\mathscr{W}$ type inferencer [8] using the same K approach. Figure 3 shows the new K annotated syntax for $\mathscr{W}$; it changes the conditional to be strict in all arguments, makes let strict in its second argument, and desugars letrec to let (because let is typed differently than its Exp desugaring). Unification over type

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

Fig. 3. K-Annotated Syntax of Exp for $\mathscr{W}$

expressions is needed and defined in Figure 4 (with $t_v \in \mathit{TypeVar}$). Fortunately, unification is straightforward to define equationally using set matching; we define it using rewrite rules, though, to emphasize that it is executable. Our definition is equivalent to the nondeterministic unification algorithm by Martelli and Montanari [30, Algorithm 1], instantiated to types and type variables, and with a particular rule evaluation order. [30, Theorem 2.3] provides a proof of correctness of the strategy. We implement their multi-set of equations by collecting equations and using associative and commutative (AC) matching. Finally, we should note that our substitution is kept canonical and is calculated as we go.

$$\begin{aligned} \mathit{Type} &::= \ldots \mid \mathit{int} \mid \mathit{bool} \mid \mathit{Type} \mapsto \mathit{Type} \mid \mathit{TypeVar} \\ \mathit{Eqn} &::= \mathit{Type} = \mathit{Type} \\ \mathit{Eqns} &::= \mathsf{Set}[\mathit{Eqn}] \end{aligned}$$

$$\begin{array}{l} (t = t) \to \cdot \\ (t_1 \mapsto t_2 = t_1' \mapsto t_2') \to (t_1 = t_1'),\ (t_2 = t_2') \\ (t = t_v) \to (t_v = t) \quad \text{when } t \notin \mathit{TypeVar} \\ t_v = t,\ t_v = t' \to t_v = t,\ t = t' \quad \text{when } t, t' \neq t_v \\ t_v = t,\ t_v' = t' \to t_v = t,\ t_v' = t'[t_v \leftarrow t] \\ \quad \text{when } t_v \neq t_v',\ t_v \neq t,\ t_v' \neq t',\ \text{and } t_v \in \mathit{vars}(t') \end{array}$$

Fig. 4. Unification

The first rule in Figure 4 eliminates non-informative type equalities. The second distributes equalities over function types to equalities over their sources and their targets; the third swaps type equalities for convenience (to always have type variables as lhs's of equalities); the fourth ensures that, eventually, no two type equalities have the same lhs variable; finally, the fifth rule canonizes the substitution. As expected, these rules take a set of type equalities and eventually produce a most general unifier for them:

Theorem 1. *Let $\gamma \in \mathit{Eqns}$ be a set of type equations. Then:*

- *The five-rule rewrite system above terminates (modulo AC); let $\theta \in \mathit{Eqns}$ be the normal form of γ.*
- *γ is unifiable iff θ contains only pairs of the form $t_v = t$, where $t_v \notin \mathit{vars}(t)$; if that is the case, then we identify θ with the implicit substitution that it comprises, that is, $\theta(t_v) = t$ when there is some type equality $t_v = t$ in θ, and $\theta(t_v) = t_v$ when there is no type equality of the form $t_v = t$ in θ.*
- *If γ is unifiable then θ is idempotent (i.e., $\theta \circ \theta = \theta$) and is a most general unifier of γ.*

Therefore, the five rules above give us a rewriting procedure for unification. The structure of θ in the second item above may be expensive to check every time the unification procedure is invoked; in our Maude implementation of the rules above, we sub-sort (once and for all) each equality of the form $t_v = t$ with $t_v \notin \mathit{vars}(t)$ to a "proper" equality, and then allow only proper equalities in the sort *Eqns* (the improper ones remain part of the "kind" [*Eqns*]). If $\gamma \in \mathit{Eqns}$ is a set of type equations and $t \in \mathit{Type}$ is some type expression, then we let $\gamma[t]$ denote $\theta(t)$; if γ is not unifiable, then $\gamma[t]$ is some error term (in the kind [*Type*]).

$$
\begin{aligned}
\mathit{KResult} &::= \mathit{Type}\\
\mathit{TEnv} &::= \mathsf{Map}[\mathit{Name}, \mathit{Type}]\\
\mathit{Type} &::= ... \mid \mathit{let}(\mathit{Type})\\
\mathit{ConfigLabel} &::= k \mid \mathit{tenv} \mid \mathit{eqns} \mid \mathit{nextType}\\
\mathit{Config} &::= \mathit{Type} \mid [\![K]\!] \mid \llparenthesis S \rrparenthesis_{\mathit{ConfigLabel}} \mid \llparenthesis \mathsf{Set}[\mathit{ConfigItem}] \rrparenthesis_{\top}\\
K &::= \ldots \mid \mathit{Type} \rightarrow K \quad [\mathit{strict}(2)]
\end{aligned}
$$

$$[\![e]\!] = \llparenthesis \llparenthesis e \rrparenthesis_k\ \llparenthesis \cdot \rrparenthesis_{\mathit{tenv}}\ \llparenthesis \cdot \rrparenthesis_{\mathit{eqns}}\ \llparenthesis t_0 \rrparenthesis_{\mathit{nextType}} \rrparenthesis_{\top}$$

$$\langle\!| \llparenthesis t \rrparenthesis_k\ \llparenthesis \gamma \rrparenthesis_{\mathit{eqns}} \rrparenthesis_{\top} = \gamma[t]$$

$$i \rightarrow \mathit{int},\ \ \mathit{true} \rightarrow \mathit{bool},\ \ \mathit{false} \rightarrow \mathit{bool}$$

$$\llparenthesis \underset{\mathit{int}}{\underline{t_1 + t_2}} \rrparenthesis_k\ \langle\!| \underset{t_1 = \mathit{int},\ t_2 = \mathit{int}}{\underline{\qquad\cdot\qquad}} \rrparenthesis_{\mathit{eqns}}$$

$$\llparenthesis \underset{(\gamma[t])[tl \leftarrow tl']}{\underline{\qquad x \qquad}} \rrparenthesis_k\ \llparenthesis \eta \rrparenthesis_{\mathit{tenv}}\ \llparenthesis \gamma \rrparenthesis_{\mathit{eqns}}\ \llparenthesis \underset{t_v + |tl|}{\underline{\quad t_v \quad}} \rrparenthesis_{\mathit{nextType}}\ \ \text{when } \eta[x] = \mathit{let}(t),\ tl = \mathit{vars}(\gamma[t]) - \mathit{vars}(\eta),\ \text{and } tl' = t_v \ldots (t_v + |tl| - 1)$$

$$\llparenthesis \underset{\eta[x]}{\underline{\ x\ }} \rrparenthesis_k\ \llparenthesis \eta \rrparenthesis_{\mathit{tenv}} \quad \text{when } \eta[x] \neq \mathit{let}(t)$$

$$\llparenthesis \underset{(t_v \rightarrow e) \curvearrowright \mathit{restore}(\eta)}{\underline{\qquad \lambda x.e \qquad}} \rrparenthesis_k\ \llparenthesis \underset{\eta[x \leftarrow t_v]}{\underline{\quad \eta \quad}} \rrparenthesis_{\mathit{tenv}}\ \llparenthesis \underset{t_v + 1}{\underline{\ t_v\ }} \rrparenthesis_{\mathit{nextType}}$$

$$\llparenthesis \underset{t_v}{\underline{t_1\ t_2}} \rrparenthesis_k\ \langle\!| \underset{t_1 = t_2 \rightarrow t_v}{\underline{\qquad\cdot\qquad}} \rrparenthesis_{\mathit{eqns}}\ \llparenthesis \underset{t_v + 1}{\underline{\ t_v\ }} \rrparenthesis_{\mathit{nextType}}$$

$$\llparenthesis \underset{e \curvearrowright ?_{=}(t_v) \curvearrowright \mathit{restore}(\eta)}{\underline{\qquad \mu x.e \qquad}} \rrparenthesis_k\ \llparenthesis \underset{\eta[x \leftarrow t_v]}{\underline{\quad \eta \quad}} \rrparenthesis_{\mathit{tenv}}\ \llparenthesis \underset{t_v + 1}{\underline{\ t_v\ }} \rrparenthesis_{\mathit{nextType}}$$

$$\llparenthesis t \rightarrow \underset{\cdot}{\underline{?_{=} t_v}} \rrparenthesis_k\ \langle\!| \underset{t_v = t}{\underline{\quad\cdot\quad}} \rrparenthesis_{\mathit{eqns}}$$

$$\llparenthesis \underset{e \curvearrowright \mathit{restore}(\eta)}{\underline{\ \mathsf{let}\ x = t\ \mathsf{in}\ e\ }} \rrparenthesis_k\ \llparenthesis \underset{\eta[x \leftarrow \mathit{let}(t)]}{\underline{\qquad \eta \qquad}} \rrparenthesis_{\mathit{tenv}}$$

$$\llparenthesis \underset{t_1}{\underline{\mathsf{if}\ t\ \ \mathsf{then}\ t_1\ \ \mathsf{else}\ t_2}} \rrparenthesis_k\ \langle\!| \underset{t = \mathit{bool},\ t_1 = t_2}{\underline{\qquad\cdot\qquad}} \rrparenthesis_{\mathit{eqns}}$$

$$\llparenthesis \underset{\cdot}{\underline{\mathit{restore}(\eta)}} \rrparenthesis_k \llparenthesis \underset{\eta}{\underline{\eta'}} \rrparenthesis_{\mathit{tenv}}$$

Fig. 5. K Configuration and Semantics of $\mathscr{W}$

Figure 5 shows the K definition of $\mathscr{W}$. The configuration has four items: the computation, the type environment, the set of type equations (constraints), and a counter for generating fresh type variables. Due to the strictness attributes, we can assume that the corresponding arguments of the language constructs (in which these constructs were defined strict) have already been "evaluated" to their types and the corresponding type constraints have been propagated. Lambda and fix-point abstractions perform normal bindings in the type environment, while the let performs a special binding, namely one to a type wrapped with a new "*let*" type construct. When names are looked up in the type environment, the "*let*" types are instantiated with fresh type variables for their "universal" type variables, namely those that do not occur in the type environment.

We believe that the K definition of $\mathscr{W}$ is as simple to understand as the original $\mathscr{W}$ procedure proposed [8] by Milner, once the reader has an understanding of the K notation. However, note that Milner's procedure is an *algorithm*, rather than a formal definition. The K definition above is an ordinary rewriting logic theory—the same as the definition of Exp itself. That does not mean that our K definition, when executed, must be slower than an actual implementation of $\mathscr{W}$. Experiments using Maude (see [31] for the complete Maude definition) show that our K definition of $\mathscr{W}$ is comparable to state of the art implementations of type inferencers in conventional functional languages: in our experiments, it was only about twice slower on average than that of OCaml, and had average times comparable, or even better than Haskell ghci and SML/NJ.

We have tested type inferencers both under normal operating conditions and under stressful conditions. For normal operating conditions, we have used small programs such as factorial computation together with small erroneous programs such as $\lambda x.(xx)$. We have built a collection of 10 such small programs and typechecked each of them 1,000 times. The results in Table 1 (average speed column) show the average time in seconds in which the type inferencer can check the type of a program. For the "stress test" we have used a program for which type inferencing is known to be exponential in the size of the input. Concretely, the program (which is polymorphic in $2^n + 1$ type variables!):

```
let $f_0 = \lambda x.\lambda y.x$ in
  let $f_1 = \lambda x.f_0(f_0 x)$ in
    let $f_2 = \lambda x.f_1(f_1 x)$ in
      ...
        let $f_n = \lambda x.f_{n-1}(f_{n-1} x)$ in $f_n$
```

takes the time shown in columns 3–7 to be type checked using OCaml (version 3.10.1), Haskell (`ghci` version 6.8.2), SML/NJ (version 110.67), our K definition executed in Maude (version 2.3), the PLT-Redex definition of the $\mathscr{W}$ procedure [4], and an "off-the-shelf" implementation of $\mathscr{W}$ using OCaml [32].

System	Average	Stress test				
	Speed	n = 10	n = 11	n = 12	n = 13	n = 14
OCaml	0.6s	0.6s	2.1s	7.9s	30.6s	120.5s
Haskell	1.2s	0.5s	0.9s	1.5s	2.5s	5.8s
SML/NJ	4.0s	5.1s	14.6s	110.2s	721.9s	-
$\mathscr{W}$ in K	1.1s	2.6s	7.8s	26.9s	103.1s	373.2s
!$\mathscr{W}$ in K	2.0s	2.6s	7.7s	26.1s	96.4s	360.4s
$\mathscr{W}$ in PLT/Redex	134.8s	>1h	-	-	-	-
$\mathscr{W}$ in OCaml	49.8s	105.9s	9m14	>1h	-	-

Table 1. Speed of various $\mathscr{W}$ implementations

These experiments have been conducted on a 3.4GHz/2GB Linux machine. All the tests performed were already in normal form, so no evaluation was necessary (other than type checking and compiling). For OCaml we have used the type mode of the Enhanced OCaml Toplevel [33] to only enable type checking. For Haskell we have used the type directive "`:t`". For SML the table presents the entire compilation time since we did not find a satisfactory way to only obtain typing time. Only the user time has been recorded. Except for SML, the user time was very close to the real time; for SML, the real time was 30% larger than the user time. Moreover, an extension to $\mathscr{W}$, which we call $!\mathscr{W}$ (see Figures 6 and 7), containing lists, products, side effects (through referencing, dereferencing, and assignment) and weak polymorphism did not add any significant slowdown. Therefore, our K *definitions* yield quite realistic *implementations* of type checkers/inferencers when executed on an efficient rewrite engine.

$$\begin{aligned}
Exp ::= {} & \ldots \mid \mathsf{ref}\, Exp \mid \texttt{\&}\ \ Var \mid \texttt{!}\ \ Exp \mid Exp\ \ \texttt{:=}\ \ Exp \mid [ExpList] \\
\mid {} & \ \mathsf{car}\, Exp \mid \mathsf{cdr}\, Exp \mid \mathsf{null?}\, Exp \mid \mathsf{cons}\, Exp\, Exp \mid Exp\ \ \texttt{;}\ \ Exp
\end{aligned}$$

$$\begin{aligned}
KResult &::= Type \\
TEnv &::= \mathsf{Map}[Name, Type] \\
ConfigLabel &::= k \mid tenv \mid eqns \mid nextType \mid results \mid mkLet \\
Config &::= Type \mid [\![K]\!] \mid (\!|S|\!)_{ConfigLabel} \mid (\!|\mathsf{Set}[Config]|\!)_{\top}
\end{aligned}$$

Fig. 6. The $!\mathscr{W}$ type inferencer, Syntax & Configuration

Our Maude "implementation" of an extension[1] to the K definition of $\mathscr{W}$ has about 30 lines of code. How is it possible that a formal definition of a type system, written in 30 lines of code, can be executed *as is* with comparable efficiency to well-engineered implementations of the same type system in widely used programming languages? We think that the answer to this question involves at least two aspects. On one hand, Maude, despite its generality, is a well-engineered rewrite engine implementing state-of-the-art AC matching and term indexing algorithms [34]. On the other hand, our K definition makes intensive use of what Maude is very good at, namely AC matching. For example, note the fourth rule in Figure 4: the type variable t_v appears twice in the lhs of the rule, once in each of the two type equalities involved. Maude will therefore need to search and then index for two type equalities in the set of type constraints which share the same type variable. Similarly, the fifth rule involves two type equalities, the second containing in its t' some occurrence of the type variable t_v that appears in the first. Without appropriate indexing to avoid rematching of rules, which is what Maude does well, such operations can be very expensive. Moreover, note that our type constraints can be "solved" incrementally (by applying the five unification rewrite

[1] With conventional arithmetic and boolean operators added for writing and testing our definition on meaningful programs.

Rules from Figure 5, with modifications or additions as follows:

$\llparenthesis\frac{x}{t'[tl \leftarrow tl']}\rangle_k \llparenthesis\eta\rrparenthesis_{tenv} \llparenthesis\gamma\rrparenthesis_{eqns} \llparenthesis\frac{t_v}{t_v + |tl|}\rrparenthesis_{nextType}$ when $\eta[x] = let(t)$, $t' = \gamma[t]$, $t' : RefType$, $tl = vars(\gamma[t]) - vars(\eta)$, and $tl' = t_v \ldots (t_v + |tl| - 1)$

$\llparenthesis\frac{\lambda xl.e}{bind(xl) \curvearrowright e \curvearrowright mkFunType(xl) \curvearrowright restore(\eta)}\rangle_k \llparenthesis\eta\rrparenthesis_{tenv}$

$t \curvearrowright mkFunType(tl) = tl \to t$

$\llparenthesis\frac{t \curvearrowright mkFunType(xl)}{\eta[xl] \to t}\rangle_k \llparenthesis\eta\rrparenthesis_{tenv}$

$\llparenthesis\frac{\mathsf{let}\, xl = el \;\mathsf{in}\; e}{el \curvearrowright mkLet(\cdot) \curvearrowright bindTo(xl) \curvearrowright e \curvearrowright restore(\eta)}\rangle_k \llparenthesis\eta\rrparenthesis_{tenv}$

$\llparenthesis\frac{t}{\cdot}\rangle_{results} \curvearrowright \langle\!|\frac{\cdot}{let\ (t)}\rrparenthesis_{mkLet}$

$\llparenthesis\frac{\cdot}{tl}\rrparenthesis_{results} \curvearrowright \frac{\llparenthesis tl\rrparenthesis_{mkLet}}{\cdot}$

$\llparenthesis\frac{\mathsf{letrec}\, xl = el \;\mathsf{in}\; e}{bind(xl) \curvearrowright el \curvearrowright addEqns(xl) \curvearrowright mkLet(\cdot) \curvearrowright bindTo(xl) \curvearrowright e \curvearrowright restore(\eta)}\rangle_k \llparenthesis\eta\rrparenthesis_{tenv}$

$\llparenthesis results(tl') \curvearrowright \frac{addEqns(tl)}{\cdot}\rangle_k \langle\!|\frac{\cdot}{tl = tl'}\rrparenthesis_{eqns}$

$results(tl) \curvearrowright addEqns(xl) = xl \curvearrowright addEqns(tl)$

$\llparenthesis\frac{[tl]}{list\ t_v}\rangle_k \llparenthesis\Gamma, \frac{\cdot}{t_v \mathrel{*}= tl}\rrparenthesis_{eqns} \llparenthesis\frac{t_v}{t_v + 1}\rrparenthesis_{nextType}$ where $\frac{t \mathrel{*}= \cdot}{\cdot}$ and $\frac{\cdot}{t_v = t}, t_v \mathrel{*}= \llparenthesis\frac{t}{\cdot}\rangle$

$\llparenthesis\frac{\mathsf{cdr}\, t}{list\ t_v}\rangle_k \llparenthesis\Gamma, \frac{\cdot}{t = list\ t_v}\rrparenthesis_{eqns} \llparenthesis\frac{t_v}{t_v + 1}\rrparenthesis_{nextType}$

$\llparenthesis\frac{\mathsf{car}\, t}{t_v}\rangle_k \llparenthesis\Gamma, \frac{\cdot}{t = list\ t_v}\rrparenthesis_{eqns} \llparenthesis\frac{t_v}{t_v + 1}\rrparenthesis_{nextType}$

$\llparenthesis\frac{\mathsf{cons}\, t_1\, t_2}{t_2}\rangle_k \llparenthesis\Gamma, \frac{\cdot}{t_2 = list\ t_1}\rrparenthesis_{eqns}$

$\llparenthesis\frac{\mathsf{null?}\, t}{bool}\rangle_k \llparenthesis\Gamma, \frac{\cdot}{t = list\ t_v}\rrparenthesis_{eqns} \llparenthesis\frac{t_v}{t_v + 1}\rrparenthesis_{nextType}$

$\llparenthesis\frac{\texttt{!}\ t}{t_v}\rangle_k \llparenthesis\Gamma, \frac{\cdot}{t = ref\ t_v}\rrparenthesis_{eqns} \llparenthesis\frac{t_v}{t_v + 1}\rrparenthesis_{nextType}$

$\texttt{\&}\ t \to ref\ t$

$\llparenthesis\frac{t_1\ \texttt{:=}\ t_2}{unit}\rangle_k \llparenthesis\Gamma, \frac{\cdot}{t_1 = ref\ t_2}\rrparenthesis_{eqns}$

$\llparenthesis\frac{t_1\ \texttt{;}\ t_2}{t_2}\rangle_k \llparenthesis\Gamma, \frac{\cdot}{t_1 = unit}\rrparenthesis_{eqns}$

Fig. 7. The $!\mathscr{W}$ type inferencer, Semantics

rules), as generated, into a most general substitution; incremental solving of the type constraints can have a significant impact on the complexity of unification as we defined it, and Maude indeed does that.

4 Analysis and Proof Technique

Here, we sketch an argument that although different in form, our definition of $\mathscr{W}$ is equivalent to Milner's. We additionally give a brief summary of our work in proving type system soundness using our formalism.

4.1 Equivalence of Milner's and Our $\mathscr{W}$

We would like to make explicit how our rewriting definition is effectively equivalent to Milner's $\mathscr{W}$, up to the additions of some explicit fundamental data types and operators. To do this, it is easiest to look at $\mathscr{J}$, Milner's simplified algorithm, which he proved equivalent to $\mathscr{W}$. Milner's definition of $\mathscr{J}$ is given in Figure 8 as a convenience for the reader. The main questions of equivalence center around recursive calls and their environments, as well as the substitution.

$\mathscr{J}(\bar{p}, f) = \tau$

1. If f is x then:
   If λx_σ is active in $\bar{p}$, $\tau := \sigma$.
   If *let* x_σ is active in $\bar{p}$, $\tau = [\beta_i/\alpha_i]E_\sigma$, where α_i are the generic type variables of *let* $x_{E\sigma}$ in $E\bar{p}$, and β_i are new variables.
2. If f is de then:
   $\rho := \mathscr{J}(\bar{p}, d)$; $\sigma := \mathscr{J}(\bar{p}, e)$;
   UNIFY$(\rho, \sigma \to \beta)$; $\tau := \beta$; (β new)
3. If f is (*if* d *then* e *else* e'), then:
   $\rho := \mathscr{J}(\bar{p}, d)$; UNIFY$(\rho, \textit{bool})$;
   $\sigma := \mathscr{J}(\bar{p}, e)$; $\sigma' := \mathscr{J}(\bar{p}, e')$;
   UNIFY(σ, σ'); $\tau := \sigma$
4. If f is $(\lambda x \cdot d)$ then:
   $\rho := \mathscr{J}(\bar{p} \cdot \lambda x_\beta, d)$; $\tau := \beta \to \rho$; (β new)
5. If f is (*fix* $x \cdot d$), then:
   $\rho := \mathscr{J}(\bar{p} \cdot \textit{fix}\ x_\beta, d)$; ($\beta$ new)
   UNIFY(β, ρ); $\tau = \beta$;
6. If f is (*let* $x = d$ *in* e) then:
   $\rho := \mathscr{J}(\bar{p}, d)$; $\sigma := \mathscr{J}(\bar{p} \cdot\ \textit{let}\ x_\rho, e)$; $\tau := \sigma$.

UNIFY is a procedure that delivers no result, but has a side effect on a global substitution E. If UNIFY(σ, τ) changes E to E', and if $\mathscr{U}(E\sigma, E\tau) = U$, then $E' = UE$, where $\mathscr{U}$ is a unification generator.

Fig. 8. Milner's $\mathscr{J}$ Algorithm

$\mathscr{J}$ is a recursive algorithm. It calls itself on subexpressions throughout the computation. We achieve the same effect through the use of strictness attributes and the saving and restoring of environments. Our strictness attributes cause subexpressions to be moved to the front of the computation structure, effectively disabling rules that would apply to the "context," and enabling rules applying to the subexpression itself.

To each call of $\mathscr{J}$, a type environment (also called a typed prefix in Milner's notation) is passed. Because we have only one global type environment, it is not immediately obvious that changes to the type environment when evaluating subexpressions cannot affect the remaining computation. In Milner's algorithm, this is handled by virtue of passing the environments by value. We ensure this by always placing, at the end of the local computation, a *restore* marker and a copy of the current environment before affecting the environment. Thus, when the local computation is complete, the environment is restored to what it was before starting the subcomputation.

Both definitions keep a single, global substitution, to which restrictions are continually added as side effects. In addition, both only apply the substitution when doing variable lookup. The calls to UNIFY in the application, if/then/else, and fix cases are reflected in our rules by the additional formulas added to the *eqns* configuration item. Indeed, for the rules of Exp (disregarding our extensions with integers), these are the only times we affect the unifier. As an example, let us look at the application de in an environment $\bar{p}$. In $\mathscr{J}$, two recursive calls to $\mathscr{J}$ are made: $\mathscr{J}(\bar{p}, d)$ and $\mathscr{J}(\bar{p}, e)$, whose results are called ρ and σ respectively. Then a restriction to the global unifier is made, equating ρ with $\sigma \to \beta$, with β being a new type variable, and finally β is returned as the type of the expression.

We do a very similar thing. The strictness attributes of the application operator force evaluation of the arguments d and e first. These eventually transform into $\rho\sigma$. We can then apply a rewrite rule where we end up with a new type β, and add an equation $\rho = \sigma \to \beta$ to the *eqns* configuration item. The evaluations of d and e are guaranteed not to change the environment because we always restore environments upon returning types.

4.2 Type Preservation

We attempted to prove the preservation property of $\mathscr{W}$ using our methodology. We briefly outline the approach below. For more details of the partial proofs of soundness for this and other type systems defined in K, see [35]. We use a few conventions to shorten statements. The variables V, E, and K stand for values, expressions, and computation items respectively. Additionally, we add $\mathscr{E}$ and $\mathscr{W}$ subscripts on constructs that are shared between both the Exp language and the $\mathscr{W}$ algorithm. We then only mention the system in which reductions are taking place if it is not immediately clear from context. Finally, a statement like $\mathscr{W} \models R \overset{*}{\longrightarrow} R'$ means that R reduces to R' under the rewrite rules for $\mathscr{W}$.

A distinguishing feature of our proof technique is that we use an abstraction function, α, to enable us to convert between a configuration in the language domain to a corresponding configuration in the typing domain. Using an abstraction function in proving soundness is a technique used frequently in the domain of processor construction, as introduced in [36], or compiler optimization [37, 38].

Lemma 1. *Any reachable configuration in the language domain can be transformed using structural equations into a unique expression.*

Proof. This follows from two key ideas. One, you cannot use the structural equations to transform an expression into any other expression, and two, each structural equation can be applied backwards even after the rules have applied.

Because of Lemma 1, we can use a simple definition for α: $\alpha(\llbracket E \rrbracket_{\mathscr{E}}) = \llbracket E \rrbracket_{\mathscr{W}}$. By the lemma, this definition is well-defined for all reachable configurations, and homomorphic with respect to structural rules. While this function is effectively the identity function, we have experimented with much more complicated abstraction functions, which lead us to believe the technique scales [35].

Lemma 2. *If* $\mathscr{W} \models \alpha(V) \xrightarrow{*} \tau$ *then* $\llbracket V \rrbracket_{\mathscr{W}} \xrightarrow{*} \tau$.

Proof. This follows directly from the $\mathscr{W}$ rewrite rules for values.

Lemma 3 (Preservation 1). *If* $\llbracket E \rrbracket_{\mathscr{W}} \xrightarrow{*} \tau$ *and* $\llbracket E \rrbracket_{\mathscr{E}} \xrightarrow{*} R$ *for some type* τ *and configuration* R*, then* $\mathscr{W} \models \alpha(R) \xrightarrow{*} \tau'$ *for some* τ' *unifiable with* τ.

Lemma 4 (Preservation 2). *If* $\llbracket E \rrbracket_{\mathscr{W}} \xrightarrow{*} \tau$ *and* $\llbracket E \rrbracket_{\mathscr{E}} \xrightarrow{*} V$ *for some type* τ *and value* V*, then* $\llbracket V \rrbracket_{\mathscr{W}} \xrightarrow{*} \tau'$ *for some type* τ'.

Proof. This follows directly from Lemmas 2 and 3.

In comparison, the definition of (strong) preservation as given by Wright and Felleisen [6, Lemma 4.3] states: “If $\Gamma \rhd e_1 : \tau$ and $e_1 \longrightarrow e_2$ then $\Gamma \rhd e_2 : \tau$.” We cannot define preservation in the same way, because our terms do not necessarily remain terms as they evaluate. If one accepts the idea of our abstraction function, then subject reduction is actually closer in spirit to the above Lemma 3. We were able to verify Lemma 3 for many of the cases, but were unable to show the necessary correspondence between the environment and replacement.

5 Conclusions and Further Work

We have shown that rewriting logic, through K, is amenable for defining feasible type inferencers for programming languages. Evaluation suggests that these equationally defined type inferencers are comparable in speed with “off-the-shelf” ones used by real implementations of programming languages. Since both the language and its type system are defined uniformly as theories in the same logic, one can use the standard RLS proof theory to prove properties about languages and type systems for those languages. These preliminary results lead us to believe our approach is a good candidate for the PoplMark Challenge [39].

WADT'08, LNCS 5486, pp 135-151. 2009

References

1. Meseguer, J., Roșu, G.: Rewriting logic semantics: From language specifications to formal analysis tools. In: IJCAR '04. Volume 3097 of LNCS., Springer (2004) 1–44
2. Meseguer, J., Roșu, G.: The rewriting logic semantics project. J. TCS **373**(3) (2007) 213–237
3. Roșu, G.: 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)
4. Kuan, G., MacQueen, D., Findler, R.B.: A rewriting semantics for type inference. In: ESOP '07. Volume 4421 of LNCS., Springer (2007) 426–440
5. Felleisen, M., Hieb, R.: A revised report on the syntactic theories of sequential control and state. J. TCS **103**(2) (1992) 235–271
6. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation **115**(1) (1994) 38–94
7. Matthews, J., Findler, R.B., Flatt, M., Felleisen, M.: A visual environment for developing context-sensitive term rewriting systems. In: RTA '04. Volume 3091 of LNCS., Springer (2004) 301–311
8. Milner, R.: A theory of type polymorphism in programming. J. Computer and System Sciences **17**(3) (1978) 348–375
9. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT Tutorial, Examples, and Reference Manual. Department of Information and Computing Sciences, Universiteit Utrecht. (August 2005) (Draft).
10. Barendregt, H.: Introduction to generalized type systems. J. Functional Programming **1**(2) (1991) 125–154
11. Stehr, M.O., Meseguer, J.: Pure type systems in rewriting logic: Specifying typed higher-order languages in a first-order logical framework. In: Essays in Memory of Ole-Johan Dahl. Volume 2635 of LNCS., Springer (2004) 334–375
12. Barendregt, H.P., van Eekelen, M.C.J.D., Glauert, J.R.W., Kennaway, R., Plasmeijer, M.J., Sleep, M.R.: Term graph rewriting. In: PARLE (2). Volume 259 of LNCS., Springer (1987) 141–158
13. Plump, D.: Term graph rewriting. In: Handbook of Graph Grammars and Computing by Graph Transformation. Volume 2. World Scientific (1998)
14. Banach, R.: Simple type inference for term graph rewriting systems. In: CTRS '92. Volume 656 of LNCS. (1992) 51–66
15. Fogarty, S., Pasalic, E., Siek, J., Taha, W.: Concoqtion: Indexed types now! In: PEPM '07, ACM (2007) 112–121
16. Kamareddine, F., Klop, J.W., eds.: Special Issue on Type Theory and Term Rewriting: A Collection of Papers. In Kamareddine, F., Klop, J.W., eds.: Journal of Logic and Computation. Volume 10(3)., Oxford University Press (2000)
17. Hünke, Y., de Moor, O.: Aiding dependent type checking with rewrite rules (2001) unpublished, http://citeseer.ist.psu.edu/huencke01aiding.html.
18. Mametjanov, A.: Types and program transformations. In: OOPSLA '07 Companion, ACM (2007) 937–938
19. Levin, M.Y., Pierce, B.C.: TinkerType: A language for playing with formal systems. J. Functional Programing **13**(2) (2003) 295–316
20. Lee, D.K., Crary, K., Harper, R.: Towards a mechanized metatheory of standard ML. In: POPL '07, ACM (2007) 173–184
21. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. TOPLAS **28**(4) (2006) 619–695

22. Sewell, P., Nardelli, F.Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strniša, R.: Ott: Effective tool support for the working semanticist. In: ICFP '07: Proceedings of the 2007 ACM SIGPLAN international conference on Functional programming, New York, NY, USA, ACM (2007) 1–12
23. van den Brand, M., Heering, J., de Jong, H., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P., Scheerder, J., Vinju, J., Visser, E., Visser, J.: The ASF+SDF Meta-Environment: A component-based language development environment. In: Proceedings of Compiler Construction 2001. LNCS, Springer (2001)
24. Roşu, G.: K: A rewriting-based framework for computations—an informal guide. Technical Report UIUCDCS-R-2007-2926, University of Illinois at Urbana-Champaign (2007)
25. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. J. TCS **96**(1) (1992) 73–155
26. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: Specification and programming in rewriting logic. Theor. Comput. Sci. **285**(2) (2002) 187–243
27. Plotkin, G.D.: A structural approach to operational semantics. Journal of Logic and Algebraic Programming **60-61** (2004) 17–139
28. Gurevich, Y.: Evolving algebras 1993: Lipari guide. In: Specification and validation methods. Oxford University Press, Inc., New York, NY, USA (1995) 9–36
29. Strachey, C., Wadsworth, C.P.: Continuations: A mathematical semantics for handling full jumps. Higher-Order and Symb. Computation **13**(1/2) (2000) 135–152
30. Martelli, A., Montanari, U.: An efficient unification algorithm. ACM Trans. Program. Lang. Syst. **4**(2) (1982) 258–282
31. Roşu, G.: K-style Maude definition of the W type inferencer (2007) http://fsl.cs.uiuc.edu/index.php/Special:WOnline.
32. Kothari, S., Caldwell, J.: Algorithm W for lambda calculus extended with Milner-let Implementation used for Type Reconstruction Algorithms—A Survey. Technical Report, University of Wyoming, 2007.
33. Li, Z.: Enhtop: A patch for an enhanced OCaml toplevel (2007) http://www.pps.jussieu.fr/∼li/software/index.html.
34. Eker, S.: Fast matching in combinations of regular equational theories. In: WRLA '96. Volume 4 of ENTCS. (1996) 90–109
35. Ellison, C.: A rewriting logic approach to defining type systems. Master's thesis, University of Illinois at Urbana-Champaign (2008) http://fsl.cs.uiuc.edu/pubs/ellison-2008-mastersthesis.pdf.
36. Hosabettu, R., Srivas, M.K., Gopalakrishnan, G.: Decomposing the proof of correctness of pipelined microprocessors. In: CAV '98. Volume 1427 of LNCS., Springer (1998) 122–134
37. Kanade, A., Sanyal, A., Khedker, U.P.: A PVS based framework for validating compiler optimizations. In: SEFM '06, IEEE Computer Society (2006) 108–117
38. Kanade, A., Sanyal, A., Khedker, U.P.: Structuring optimizing transformations and proving them sound. In: COCV '06. Volume 176(3) of ENTCS., Elsevier (2007) 79–95
39. Aydemir, B.E., Bohannon, A., Fairbairn, M., Foster, J.N., Pierce, B.C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., Zdancewic, S.: Mechanized metatheory for the masses: The PoplMark challenge. In: TPHOLs '05. Volume 3603 of LNCS., Springer (2005) 50–65