On Compiling Rewriting Logic Language Definitions into Competitive Interpreters

Michael Ilseman Chucky Ellison Grigore Rosu
University of Illinois at Urbana-Champaign
ilseman2@illinois.edu/celliso2@illinois.edu/grosu@illinois.edu

Abstract
This paper describes a completely automated method for generating efficient and competitive interpreters from formal semantics expressed in Rewriting Logic. The semantics are compiled into OCaml code, which then acts as the interpreter for the language being defined. This automatic translation is tested on the semantics of an imperative as well as a functional language, and these generated interpreters are then benchmarked across a number of programs. In all cases the compiled interpreter is faster than directly executing the definition in a Rewriting system with improvements of several orders of magnitude.

Keywords K, Rewriting Logic, formal semantics, interpreters

1. Introduction
Formal programming language semantics have been around almost as long as programming languages themselves. Numerous formalisms have been introduced, with differing strengths and weaknesses, yet most programming language development is still done informally. While there are likely many reasons why this is the case, one of the simplest is that people want interactive development—they want immediate feedback while they work on their design.

Our primary goal is to make working with formally defined programming languages easier than working with only implementations or natural language specifications. Although programmers are more familiar with simply writing compilers or interpreters based on informal specifications, the actual language in which semantics are expressed is only a small part of the entire package. If the only computer-readable “definition” is a compiler or interpreter, then the programmer still has to write debuggers, integrated development environments, refactoring tools, type checkers, model checkers, verification tools, etc.

Instead, if your language definition is a formal, mathematical description, computers can actually generate these secondary tools, or at the very least assist in generating them. This is because a formal definition can be easily analyzed and transformed. Using our particular language formalism, called the K Framework, we already generate a number of the above secondary tools. Most importantly, such semantic definitions are directly executable as interpreters in a rewriting system such as Maude [5, 6]. However, previous work has shown these interpreters are 10 to 100 times slower than bc (calculator language) implementations and around 10000 times slower than pure C [11]. While these speeds are fast enough for development purposes, they are not nearly fast enough for general purpose use. This paper seeks to extend the previous effort in order to generate interpreters that are potentially feasible for end-user use. If the generated interpreters are sufficiently fast, it may relieve the designer entirely of the burden of implementation.

In Sec. 2 we give some information useful in understanding the underlying semantics of our methodology. Section 4 describes the actual transformation process we use, which is followed by an example transformation in Sec. 5. The system is then evaluated in Sec. 6. We finish in Sec. 7 with some comparisons to similar work as well as ideas for future work.

2. Background
Because we use Rewriting Logic in the K Framework to express our language semantics, here we give a quick review of both. The role of this section is to establish concepts and notations used later in the paper.

The K Framework [19] is a methodology and tool-set allowing one to formally specify the semantics of a programming language in an inherently executable and modular way. The fundamental logic providing the mathematical meaning of language definitions in the K Framework is Rewriting Logic (RL). RL [16], not to be confused with context reduction or term rewriting, organizes term rewriting modulo equations as a logic with a complete proof system and initial model semantics. To be explicit, K is an extended subset of RL. It is a subset of RL in the sense that it suggests certain stylistic conventions to be adopted by language designers in order to implement their languages. This restriction streamlines the logic in order to offer pre-built language modules, to make definitions more consistent, and to make them more modular.

At the heart of rewriting logic are rewriting rules. Any time the left-hand side (LHS) of a rule is able to match a part of the configuration, the rule applies and the subterm is transformed based on the rule. Each side is allowed to contain variables, although variables on the RHS must appear on the LHS. Rules are written in a number of general styles. Structural rules are internal rules used for massaging the form of the configuration. Structural rules are written \( LHS \rightarrow RHS \). We use the equality symbol to suggest that configurations resulting from the application of structural rules are actually identical, or at least in the same equivalence class for the sake of any reasoning. Semantic rules are the rules that do actual work. Semantic rules are written \( LHS \\rightarrow RHS \).

In K, these rules work over a configuration consisting primarily of multi-sets and lists. These collections, defined algebraically in the underlying logic, contain pieces of the program state like code fragments to be executed, environments, threads, locks, etc. We think of the pieces of the configuration as floating in a solution (an associative/commutative “soup”) and so the previously mentioned rules and equations can match and change any of the pieces of the configuration it needs, giving rise to modular language definitions. The individual pieces of the configuration are called cells, and are (not necessarily uniquely) named, so rules can match certain pieces explicitly.

In a number of respects K is similar to other semantics languages such as Modular Structural Operational Semantics (MSOS) [18], Chemical Abstract Machine (CHAM) [3], or Context Reduc-
We now describe the process we used to automatically transform the Maude rewrite engine into OCaml. This involves two main steps: the definition of the language semantics and the rewriting rules, followed by the generation of the OCaml code.

3. Language Definitions

K-definitions of language semantics consist of two parts—a description of program configurations and a list of rewriting rules. As mentioned in Section 2 the configurations are multi-sets which contain pieces of the program state. The rewriting rules specify a computational step in the language. Only the relevant parts of the configuration need to be mentioned in the rewriting rules, and this lends modularity to the K-definitions.

Here we show the definitions of two simple languages using the K framework. Figure 1 shows the definition of IMP, a simple imperative language with assignment, if, and while statements. Numerically, it supports common arithmetic operations over arbitrarily sized integers.

\[
\langle \ldots \rangle_k \langle \ldots \rangle_{\text{inc}}_T \langle \ldots \rangle_{\text{result}}
\]

is the configuration of IMP. IMP contains a \(k\) cell, which contains the program instructions, a \(\text{state}\) cell, which is a global state for the language, and a \(\text{result}\) cell which will hold the result of the computation upon termination. The rewrite rules give the operations of the language.

Figure 2 shows the definition of FUN, a simple functional language that supports recursion, references, and higher-order functions.

\[
\langle \ldots \rangle_k \langle \ldots \rangle_{\text{env}} \langle \ldots \rangle_{\text{store}} \langle \ldots \rangle_{\text{nextLoc}}_T \langle \ldots \rangle_{\text{result}}
\]

is the configuration for FUN. The \(\text{env}\) cell contains the local environment inside a function, and \(\text{nextLoc}\) contains a counter for getting fresh memory locations (for use by references). The locations are in the \(\text{store}\) cell, which is the program’s heap.

4. Methodology

We now describe the process we used to automatically transform the K formal executable semantics of languages into OCaml interpreters.

4.1 Overview

A high-level view of the “OCamlization” process can be seen in Fig. 3. The generation of interpreters is shown by the vertical lines, and the compilation of individual programs expressed in the defined language is shown on the horizontal.

Interpreters are generated by passing the syntax and the semantics of the language into the K-Prototype front-end. The K-Prototype front-end parses the semantic rules, and generates a Maude-based intermediate representation (MIR). The MIR encaptulates both the semantic rules of the language as well as any strictness (evaluation order) constraints. This is where the interpreter generation process feeds into our back-end OCaml code defining an interpreter for the specified language is output from our OCAMLIZER back-end and fed into OCaml, which produces a native binary that programs can be linked to.

After the matches for all K rules have been generated, native match statements (e.g. for integer + or boolean and) are created and eval is finished. The last step is to output the footer, or a collection of utility functions for executing programs and viewing their results (e.g. print_result).

4.2 Using Rewriting for Compilation

Maude, our rewriting engine of choice, proved very adept at undertaking this transformation and generation. The facilities of meta-Maude allowed for easy term-traversal, and every term could be rewritten into a string expressing OCaml code. Rewriting is a very natural way to undertake the process of translation between languages, and with some simple string and meta utility modules we were able to focus more effort on how to translate terms into OCaml and less on the logistics and scaffolding.

4.3 Difficulties, Barriers, and Solutions

Many differences exist between the capabilities of Maude and OCaml, and these differences make translation between a Maude-based intermediate language and OCaml difficult. The main issues come from Maude’s expressiveness in the realm of subtyping, matching, rewrite-anywhere capabilities, and associativity.

4.3.1 Type Hierarchies

Due to this structure, and the generality of convert, programs written in the defined language are simply passed through convert where they emerge as OCaml expressions ready to be linked with the language interpreter and compiled into machine code executables.

4.4 Using Rewriting for Compilation

Maude, our rewriting engine of choice, proved very adept at undertaking this transformation and generation. The facilities of meta-Maude allowed for easy term-traversal, and every term could be rewritten into a string expressing OCaml code. Rewriting is a very natural way to undertake the process of translation between languages, and with some simple string and meta utility modules we were able to focus more effort on how to translate terms into OCaml and less on the logistics and scaffolding.

4.4.1 Type Hierarchies

The intuition for the generated type layout in OCaml is to divide computation into two main categories: Results and Causes. Results are computations that are completed, yielding a result that can then be used. Thus strictness can be satisfied by matching against a Result. Causes are computations that are not yet completed, and if they appear in the place of a strict argument, they are then to be evaluated until they become Results. K configurations are implemented via tuples, K cells are implemented as lists, and cells that define mappings are implemented as hash-tables.

The following is the general type system layout:

\[
\text{type } k = \text{Cause of cause} | \text{Result of result} | \text{Hash of (k, k)} | \text{Hashtbl of t} | \text{Empty} |
\text{and result = int of int} | \text{bool of bool} | \text{BigInt of big_int} | \text{ResultList of result list} | \text{NORESULT} |
\text{and cause = Apply of k list} | \text{KList of k list} | \text{Results} | \text{Variable of string} | \text{CauseList of cause list} |
\text{... <generated constructors> ...}
\]
which is very expressive and powerful. Lists in Maude are associative and can be matched against as such. Associative matching means that lists, e.g. with the “,” separator operator, can be matched as (L1, L2), where L1 and L2 are themselves lists, in addition to (E, L1) where E is an element.

The OCaml equivalent to “,” is the list append operator “@”, but this cannot be matched against. The only operator that can be matched against is cons, “::”, which has type ’a -> ’a list -> ’a list, meaning that it can only match an element onto a list. Thus OCaml list matching is more analogous to matching a stack — to match an element in the stack, all prior elements must be syntactically “popped” off and themselves matched. One of the benefits of the formalism used in this paper is that the k cells are themselves usually matched at the top for the majority of language constructs. This is because an attribute of executable semantics is that execution follows specification, and the intuitive understanding of execution is that it proceeds in a task to task, expression to expression manner. While the K-Prototype implements the full functionality of associative matching, encouraging a more stack based execution model on a language does not usually, in practice, restrict or run contrary to intuition or intent. This means that language behavior is defined by what is at the top of the computation stack, and that the top of this stack is rewritten into another expression that is on the top of the rest of the computation to be performed.

4.3.3 Templates for K configurations

Rewriting Logic and K allow the designer to only specify the relevant parts of the configuration in a K rule. This is handled in OCAMLIZER by the generation of a template that explicitly matches and deconstructs the K configuration for the language being used. The generated template for a language is created by parsing the language configuration and creating an OCaml tuple that fully expresses its structure.

This template consists of valid OCaml code with certain place-
holders, or “bubbles” into which converted code can be inserted. Since the template itself is valid code, only some of the bubbles need to be filled in, the others will just be matched against as variables. The bubbles are of the form “ooCELLoo” where CELL is the name of the cell in the K configuration that resides in its location. The intuition is that the surrounding “oo” marks that it is a template place-holding bubble for a cell to be plugged into. More importantly, a bubble that is not filled in functions as a variable for OCaml to match against when a K rule has no need of that particular cell. This implements much of the match and rewrite-anywhere capabilities that are relevant to the K configurations. This process is demonstrated in the example in Sec. 5.

4.3.4 Names

The set of names that OCaml can use is much more limited than Maude, so a simple translation on names is done. We call this process legalization. This process must be sensitive to whether a given name is a type constructor or variable name because OCaml requires the first to be capitalized and the second to be lower case. Legalization is done by translating any non-letter ascii character into some letter-based representation. For example, ‘~’ would be translated to the legal constructor name TildeEqTilde, and E would become the variable vE, etc. In a configuration file, the language designer can optionally specify any specific legalizations that he may prefer over the automatically generated ones. This can lead to more readable generated code, especially for common constructs. Some default legalizations we have defined to increase readability are for ‘_->_’ and ‘_‘(‘_’)’ be legalized to KList and Apply instead of the UnderMinGtUnder and UnderTicLpUnderTicRp that would normally be generated.

4.4 Additional Difficulties

The rule for lookup in one of the languages we worked with, FUN, is defined as

\[
\text{FUN-LOOKUP: } \langle x \overset{\text{k}}{\to} k \rangle \kappa \langle \rho \rangle \epsilon \sigma \pl \\
\rightarrow \langle \sigma \rangle \epsilon \langle \rho \rangle \epsilon \sigma \pl \langle \rho \rangle \epsilon \langle \sigma \rangle \pl \rangle
\]

The cell layout is associative-commutative, yet matching is simple as the cells are not matched against one another. Furthermore there is an ordering specified to how the computation proceeds — \( x \) is matched, then it is looked up in \( \rho \), and that result is looked up in \( \sigma \). However, this is not the only way one may define variable lookup. Below we show an alternative definition for lookup for FUN that is more difficult to translate generically to OCaml:

\[
\text{"HARDER" LOOKUP: } \langle x \overset{\text{k}}{\to} L \rangle \kappa \langle \rho \rangle \epsilon \langle \sigma \rangle \pl \langle \rho \rangle \epsilon \langle \sigma \rangle \pl \rangle
\]

The cell layout here is also associative-commutative, but now the contents of the cells are matched against one another. This means that there is no explicit order or procession of computation. Conceivably, an implementation could match against some \( l \), and given its corresponding \( L \) traverse \( \rho \) to find an \( x \), continue to the K cell only to reject its match and start over with a different \( l \). This could be done for every \( l \) in \( \sigma \) until either a match is found and the rule applies, or a match is not found and the rule does not apply. While this may still be a correct implementation, such a performance penalty is unacceptable.

An approach to handling this problem is to define a partial ordering on the types of matches involved, where a match is more
5. Example

For this example we will use the language FUN (see Appendix Figure 2). The first example will show how the equation defining letrec is converted into a match statement for the eval function to call. The second example converts a simple program into an OCaml program that uses the generated interpreter to interpret itself.

5.1 Letrec

Here is an example transformation of the rule for letrec. The rule is defined as:

\[
\text{letrec } \cdots \text{ in } \cdots \text{ end letrec}
\]

This is turned into the following meta-term by the K-Prototype tool:

\[
\begin{align*}
\text{letrec } \cdots \text{ in } \cdots \text{ end letrec} & \Rightarrow \text{letrec } \cdots \text{ in } \cdots \text{ end letrec} \\
\text{letrec } \cdots \text{ in } \cdots \text{ end letrec} & \Rightarrow \text{letrec } \cdots \text{ in } \cdots \text{ end letrec} \\
\text{letrec } \cdots \text{ in } \cdots \text{ end letrec} & \Rightarrow \text{letrec } \cdots \text{ in } \cdots \text{ end letrec}
\end{align*}
\]

The meta-terms are all in a meta prefix notation and handled in that form, but for the purpose of the example and readability, we present such terms in a "pretty printed" form. The above can be more easily read (still in prefix-form) as:

\[
\begin{align*}
\text{letrec } \cdots \text{ in } \cdots \text{ end letrec} & \Rightarrow \text{letrec } \cdots \text{ in } \cdots \text{ end letrec} \\
\text{letrec } \cdots \text{ in } \cdots \text{ end letrec} & \Rightarrow \text{letrec } \cdots \text{ in } \cdots \text{ end letrec} \\
\text{letrec } \cdots \text{ in } \cdots \text{ end letrec} & \Rightarrow \text{letrec } \cdots \text{ in } \cdots \text{ end letrec}
\end{align*}
\]

We will use this somewhat nicer notation for the rest of the paper. The part of the conversion process that constructs eval takes all semantic equations and translates them into match statements in OCaml. The construction of a match statement looks like:

\[
\begin{align*}
\text{letrec } \cdots \text{ in } \cdots \text{ end letrec} & \Rightarrow \text{letrec } \cdots \text{ in } \cdots \text{ end letrec} \\
\text{letrec } \cdots \text{ in } \cdots \text{ end letrec} & \Rightarrow \text{letrec } \cdots \text{ in } \cdots \text{ end letrec} \\
\text{letrec } \cdots \text{ in } \cdots \text{ end letrec} & \Rightarrow \text{letrec } \cdots \text{ in } \cdots \text{ end letrec}
\end{align*}
\]

where LHS is the left-hand side of the equation, and RHS the right.

The function plug scans a language definition and generates a template representing the configuration (see Templates in Sec. 4.3.3). It then "plugs" its second argument into the fields specified by its first via insertion into the template.

The configuration for FUN is:

\[
\begin{align*}
\text{letrec } \cdots \text{ in } \cdots \text{ end letrec} & \Rightarrow \text{letrec } \cdots \text{ in } \cdots \text{ end letrec} \\
\text{letrec } \cdots \text{ in } \cdots \text{ end letrec} & \Rightarrow \text{letrec } \cdots \text{ in } \cdots \text{ end letrec} \\
\text{letrec } \cdots \text{ in } \cdots \text{ end letrec} & \Rightarrow \text{letrec } \cdots \text{ in } \cdots \text{ end letrec}
\end{align*}
\]

The function plug sees that \(\langle \text{KName}\rangle\) is a parent cell, and enters a nested tuple. \(\langle \text{KName}\rangle\), \(\langle \text{KInt}\rangle\), \(\langle \text{KInt}\rangle\), \(\langle \text{KInt}\rangle\), \(\langle \text{KInt}\rangle\), etc.

The term letPresentCellNames traverses its argument and extracts the used cell names, while mkConstructs extracts the terms from the cells, converts them into OCaml equivalents, and returns them for plug to insert into its template.

The left-hand side K term is extracted, and processed into an OCaml term to be matched against. The contents of the cell are:

\[
\text{letrec } \cdots \text{ in } \cdots \text{ end letrec}
\]

The translation is done by passing this term to convert, which traverses the term and builds up an OCaml equivalent. The \(\langle \text{KName}\rangle\), \(\langle \text{KInt}\rangle\), \(\langle \text{KInt}\rangle\), etc. Legalization proceeds as described in Sec. 4.3.4 under Names. The whole left-hand side K term ends up being translated as:

\[
\begin{align*}
\text{letrec } \cdots \text{ in } \cdots \text{ end letrec} & \Rightarrow \text{letrec } \cdots \text{ in } \cdots \text{ end letrec} \\
\text{letrec } \cdots \text{ in } \cdots \text{ end letrec} & \Rightarrow \text{letrec } \cdots \text{ in } \cdots \text{ end letrec} \\
\text{letrec } \cdots \text{ in } \cdots \text{ end letrec} & \Rightarrow \text{letrec } \cdots \text{ in } \cdots \text{ end letrec}
\end{align*}
\]

which OCaml can match against. Translations are similarly done for the other cells, and thus, the final match statement is:

\[
\begin{align*}
\text{letrec } \cdots \text{ in } \cdots \text{ end letrec} & \Rightarrow \text{letrec } \cdots \text{ in } \cdots \text{ end letrec} \\
\text{letrec } \cdots \text{ in } \cdots \text{ end letrec} & \Rightarrow \text{letrec } \cdots \text{ in } \cdots \text{ end letrec} \\
\text{letrec } \cdots \text{ in } \cdots \text{ end letrec} & \Rightarrow \text{letrec } \cdots \text{ in } \cdots \text{ end letrec}
\end{align*}
\]

5.2 Factorial

The Definition and an application of factorial in FUN is shown below:

\[
\begin{align*}
\text{letrec } f & = \text{fun } x \rightarrow \begin{cases} 1 & \text{if } (x < 1) \\ x \times (f (x - 1)) & \text{else} \end{cases} \\
\text{in } f & 5
\end{align*}
\]

The K-Prototype front-end parses this into the following meta-term:

\[
\begin{align*}
\text{letrec } \cdots \text{ in } \cdots \text{ end letrec} & \Rightarrow \text{letrec } \cdots \text{ in } \cdots \text{ end letrec} \\
\text{letrec } \cdots \text{ in } \cdots \text{ end letrec} & \Rightarrow \text{letrec } \cdots \text{ in } \cdots \text{ end letrec} \\
\text{letrec } \cdots \text{ in } \cdots \text{ end letrec} & \Rightarrow \text{letrec } \cdots \text{ in } \cdots \text{ end letrec}
\end{align*}
\]

\[
\begin{align*}
\text{convert } \text{letrec } f & = \text{fun } x \rightarrow \begin{cases} 1 & \text{if } (x < 1) \\ x \times (f (x - 1)) & \text{else} \end{cases} \\
\text{in } f & 5
\end{align*}
\]

\[
\begin{align*}
\text{convert } \text{letrec } f & = \text{fun } x \rightarrow \begin{cases} 1 & \text{if } (x < 1) \\ x \times (f (x - 1)) & \text{else} \end{cases} \\
\text{in } f & 5
\end{align*}
\]
6. Evaluation

Here we describe the experiments we made with our system. First we briefly describe the two languages we tried compiling, then the specific benchmarks and comparisons with other interpreted languages.

6.1 Languages

We worked with two languages during the creation and evaluation of our system. The first, IMP, is a simple imperative language with assignment, if, and while statements. The second, FUN, is a simple functional language that supports recursion, references, and higher-order functions. Both languages support arbitrarily sized integers. The curious reader should see Appendix Figures 1 and 2 for full definitions.

6.2 Benchmarks

For the benchmarking we used a system with 2 CPUs running at 2.53GHz with 4GB memory. The versions of our software were as follows: Maude 2.4, OCaml 3.12.0, GNU bc 1.06, Ruby 1.8.7, and Python 2.6.5. Each benchmark was averaged over at least five non-consecutive runs. The benchmark programs are available for download on our website [12] and in the appendix.

For the benchmarks, we defined a number of programs in IMP and FUN, and implemented their equivalents in Ruby, Python, and GNU bc. The aim of the benchmarks is to test how well K-Maude can handle the K language definitions running using the rewriting engine Maude.

For the iterative Fibonacci program Fib for IMP, Ruby, K-OCaml, and Python were similar in performance, with bc and K-Maude lagging behind considerably. Of interest here is that the generated K-OCaml interpreter outperformed bc, and kept on par with Python and Ruby. For the iterative factorial program Fact for IMP, K-OCaml greatly outperformed the other languages. Interestingly here is that in contrast to the majority of the benchmarks, K-Maude scaled better than the other three languages. For the iterative program Sum for IMP, K-Maude essentially was unable to scale, with an almost vertical slope. K-OCaml was a dramatic improvement, but still performed worse than the other three languages, though its slope was more in line with the other languages than with K-Maude. For the Collatz program for IMP, which proves the Collatz conjecture for all values up to its input, K-Maude again was unable to properly scale. K-OCaml again performed between K-Maude and the other languages.

For the exponential Fib function for FUN, every language was able to operate reasonably for some time before hitting a point where the exponential nature of the problem caused execution time to grow unreasonably high. K-Maude hit this point much earlier than all the other languages, and K-OCaml hit this point before the other three, but closer to Ruby than to K-Maude. The reason for this performance is that function calls are simulated in the generated interpreters through a generic process that is agnostic of what constructs in the language are for functions. The other three languages, in contrast, are hand-coded and have the luxury of knowing their own constructs and thus can implement function calls with less overhead. The recursive Fact function for FUN, like it was for IMP, showed K-OCaml performing best out of all the languages, and dramatically better than K-Maude (which in contrast to the iterative version, is essentially unable to scale). The recursive Sum for FUN showed one of the worst relative performances of K-Maude, with a nearly vertical slope. K-OCaml was among the other languages in performance, with Ruby being slowest among them. The Hanoi program for FUN, another exponential function similar to Fib, showed similar results, with all languages eventually fitting an exponential pattern. However, this time K-OCaml was a little worse at scaling in comparison to how it performed in Fib.

In every benchmark, K-OCaml out-performed K-Maude, and in many of these by orders of magnitude. The runtime profile of these programs under K-OCaml make these generated interpreters a much more viable option for end-users than the one provided by K-Maude. An issue that came up with K-OCaml in the exponentially recursive problems was high memory usage as a result of environment copying and closure passing. These benchmarks revealed to us many potential areas for optimizations, and these are discussed in Sec. 7.2.

7. Conclusions

We have demonstrated a mechanism through which programming language definitions written using the K framework can be compiled into OCaml interpreters for that defined language. The OCaml interpreters offer dramatic speed advantages compared to the same definitions running using the rewriting engine Maude.

7.1 Related Work

Andrews et al. [1] describes an interpreter derived from a formal definition of Modula-2, but we could not find any evidence that they completed work on the proposed algorithms to automatically generate the interpreter.

CENTAUR [4] is another older system that can generate interpreters from formal specifications of a language. They experimented with using both ASF [2] and “Natural Semantics” [13] (big-step SOS). Although big-step definitions lend themselves to executability, they lack many other features useful in a definitional framework such as modularity or concurrence, which K handles naturally.
Figure 4. Benchmark results. X-axis represents input number, Y-axis is time in seconds.
The ASF+SDF Meta-Environment [8], a successor to CEN- TAU R, supports the ASF+SDF Compiler [25]. This compiler can translate specifications written in the ASF+SDF framework to C code which can then be compiled and run natively. Most of the current work of the project seems to be focused on program analysis and transformation, but because they allow semantics to be written as term-rewriting, they have many of the advantages of K. Their framework does not support full matching modulo commutivity, which means it is difficult to ignore order of program state. This, in turn, may lead to less modular definitions. Additionally, ASF+SDF has no concept of rules as in Rewriting Logic, so there is no natural way to represent that certain operations are concurrent.

The LISA system [15] can generate compilers and interpreters (as well as a number of other useful tools) from finite state automata and attribute grammar descriptions of programming languages. However, it appears that their formal specification language is fairly limited — while the attribute grammars can be used to specify some simple semantic constructs, any moderately difficult construct (assignment, conditionals, etc.) is specified with Java. With this in mind, it is understandable that they are able to execute specifications but also raises questions about the formality of much of their semantics.

7.2 Future Work

In the future we would like to expand on this work by allowing for a wider variety of K constructs — most importantly, the ability to apply rules to any arbitrary location in a K cell as well as some form of AC matching.

We would also like to explore compiling the semantics to languages other than OCaml, such as Haskell or Erlang. Haskell also offers similar matching capabilities to OCaml, so the translation should follow similarly. It would be interesting to see in which of the two languages the generated code runs faster, and whether Haskell’s laziness and newer optimizations can provide interesting improvements. Erlang would be interesting to explore the highly-concurrent nature of K definitions. Aside from functional languages, compiling K definitions to a language like C may offer a tremendous speed increase. The ASF+SDF compiler can transform their rewriting-based definitions to C using a C back-end called AFerm [24], a library for term manipulation and storage. It is possible we may be able to use this package for our own framework.

Special care may be needed when compiling languages that are inherently concurrent. The K framework allows notations to be added to equations involved in concurrent rewrites, so it is possible to apply this information in the interpreter. Ideally threads in the defined language could be translated to threads in the target language. At the very least, even a non-generic solution could offer tremendous benefits when defining languages that rely on concurrency to be efficient.

Finally, we would like to explore incorporating some of the optimizations inspired from the results of the benchmarks. Memory usage and function call overhead could be lowered in K-OCaml by having a shared structure for environments, where copies are only performed on values that are overwritten rather than copying the entire environment up front. Another potential optimization would be the recognition of tail recursion and performing optimizations to discard the unneeded environments. Most of the values in the OCaml interpreters are fully constructed into terms that fit the type hierarchy. It may be possible to apply something analogous to autoboxing in Object Oriented languages to the generated OCaml code. Finally, it could be profitable to detect when a cell that defines a mapping is indexed by integers, and use a vector for the implementation rather than a hash-table.

References


**Figure 5. Imperative Fib**

**IMP:**

```imp
x := 1;
n := 1;
y := 1;
z := 1;
while ( n <= input ) (  
t := y;
x := y;
y := z;
z := t + y;
n := n + 1 );
x + 0
```

**bc:**

```bc
b = read() ;  
x = 1 ;  
y = 1 ;  
z = 1 ;  
n = 1 ;  
while ( n <= b ) {  
t = y ;  
x = y ;  
y = z ;  
z = t + y ;  
n = n + 1 } ;  
print x  
quit
```

**Python:**

```python
import sys
b = int(sys.stdin.readline())
x = 1
y = 1
z = 1
n = 1
while ( n <= b ):
  t = y
  x = y
  y = z
  z = t + y
  n = n + 1
print x
```

**Ruby:**

```ruby
b = STDIN.gets.to_i
x = 1
y = 1
z = 1
n = 1
while ( n <= b )
  t = y
  x = y
  y = z
  z = t + y
  n = n + 1
end
puts x
```

**Figure 6. Imperative Fact**

**IMP:**

```imp
x := 1;
n := 1;
while ( n <= input ) (  
x := x * n ;  
n := n + 1 );
x + 0
```

**bc:**

```bc
b = read() ;  
x = 1 ;  
n = 1 ;  
while ( n <= b ) {  
x = x * n ;  
n = n + 1 } ;  
print x  
quit
```

**Python:**

```python
import sys
b = int(sys.stdin.readline())
x = 1
n = 1
while ( n <= b ):
  x = x * n
  n = n + 1
print x
```

**Ruby:**

```ruby
b = STDIN.gets.to_i
x = 1
n = 1
while ( n <= b )
  x = x * n
  n = n + 1
end
puts x
```
**IMP:**

\[
\begin{align*}
x & := 1 ; \\
n & := 1 ; \\
\text{while} ( n \leq \text{input} ) ( \\
& \quad x := x + n ; \\
& \quad n := n + 1 ) ; \\
x & = 0
\end{align*}
\]

**Python:**

```python
import sys
b = int(sys.stdin.readline())
x = 1
n = 1
while ( n <= b ):
    x = x + n
    n = n + 1
print x
```

**Ruby:**

```ruby
b = STDIN.gets.to_i
x = 1
n = 1
while ( n <= b )
    x = x + n
    n = n + 1
end
puts x
```

**bc:**

\[
\begin{align*}
b & = \text{read()} ; \\
x & = 1 ; \\
n & = 1 ; \\
\text{while} ( n \leq b ) \{ \\
& \quad x = x + n ; \\
& \quad n = n + 1 \} ; \\
\text{print } x \\
\text{quit}
\end{align*}
\]
**Figure 8. Imperative Collatz**

**IMP:**

```
steps := 0 ; nr := 2 ;
while (nr <= input) (  
    n := nr ; nr := nr + 1 ;
    while (not(n <= 1)) (  
        steps := steps + 1 ;
        d := 0 ;
        nn := 2 ;
        while (nn <= n) (  
            d := d + 1 ;
            nn := nn + 2  
        ) ;
        if (nn <= (n + 1))  
            {n := 3 * n + 1}  
        else {n := d}
    ) ;
    steps
)
```

**Python:**

```
import sys
b = int(sys.stdin.readline())
steps = 0
nr = 2
while (nr <= b):
    n = nr
    nr = nr + 1
    while (not(n <= 1)):
        steps = steps + 1
        d = 0
        nn = 2
        while (nn <= n):
            d = d + 1
            nn = nn + 2
        if (nn <= (n + 1)):
            n = 3 * n + 1
        else:
            n = d
    steps
print steps
```

**Ruby:**

```
b = STDIN.gets.to_i
steps = 0
nr = 2
while (nr <= b)
    n = nr
    nr = nr + 1
    while (not(n <= 1)):
        steps = steps + 1
        d = 0
        nn = 2
        while (nn <= n):
            d = d + 1
            nn = nn + 2
        end
        if (nn <= (n + 1)):
            n = 3 * n + 1
        else:
            n = d
        end
    end
puts steps
```
Figure 9. Functional Fib

bc:

```bc
b = read() ;
define fib(n)
{
  if (n <= 2) {return 1;}
  else {return fib(n-1) + fib(n-2);}
}
print fib(b);
quit;
```

Python:

```python
import sys
b = int(sys.stdin.readline())
def fib(n):
  if (n <= 2):
    return 1
  else:
    return fib(n-1) + fib(n-2)
print fib(b)
```

Ruby:

```ruby
b = STDIN.gets.to_i
def fib(n)
  (n <= 1) ? 1 : n * fib(n-1)
end
puts fib(b)
```

Figure 10. Functional Fact

bc:

```bc
b = read() ;
define fact(n)
{
  if (n <= 1) return 1;
  return n * fact(n-1);
}
print fact(b);
quit;
```

Python:

```python
import sys
b = int(sys.stdin.readline())
sys.setrecursionlimit(1000000000)
def f(n):
  if (n <= 1):
    return 1
  else:
    return n * f(n-1)
print f(b)
```

Ruby:

```ruby
b = STDIN.gets.to_i
def f(n)
  (n <= 1) ? 1 : n * f(n-1)
end
puts f(b)
```
**Figure 11. Functional Sum**

**bc:**

```plaintext
b = read() ;
define fact(n)
{
    if (n <= 1) return 1;
    return n + fact(n-1);
}
print fact(b);
quit;
```

**Python:**

```python
import sys
b = int(sys.stdin.readline())
sys.setrecursionlimit(1000000000)
def f(n):
    if (n <= 1):
        return 1
    else:
        return n + f(n-1)
print f(b)
```

**Ruby:**

```ruby
b = STDIN.gets.to_i
def f(n)
    (n <= 1) ? 1 : n + f(n-1)
end
puts f(b)
```

---

**Figure 12. Functional Hanoi**

**bc:**

```plaintext
define solve(n,src,aux,dst)
{
    if (n == 0) return
    z = solve(n-1, src, dst, aux)
    z = solve(n-1, aux, src, dst)
}
n = read() ;
z = solve(n,0,2,1) ;
quit
```

**Python:**

```python
import sys
b = int(sys.stdin.readline())
global z
def solve(n,src,aux,dst):
    if (n == 0):
        return
    z = solve(n-1,src,dst,aux)
    z = solve(n-1,aux,src,dst)
z = solve(b,0,2,1)
print z
```

**Ruby:**

```ruby
b = STDIN.gets.to_i
def solve(n,src,aux,dst)
    if (n == 0)
        return
    end
    z = solve(n-1,src,dst,aux)
    z = solve(n-1,aux,src,dst)
end
z = solve(b,0,2,1)
print z
```