University of Illinois at Urbana-Champaign
Department of Computer Science Technical Report UIUCDCS-R-2005-2667

A Rewrite Framework for Language Definitions and for Generation of Efficient Interpreters ⋆

Mark Hills [a,*] Traian Şerbănuţă [a] Grigore Roşu [a]

[a] *Department of Computer Science*
University of Illinois at Urbana-Champaign, USA
201 N Goodwin Ave, Urbana, IL 61801

Abstract

A rewrite logic semantic definitional framework for programming languages is introduced, called K, together with partially automated translations of K language definitions into rewriting logic and into C. The framework is exemplified by defining SILF, a simple imperative language with functions. The translation of K definitions into rewriting logic enables the use of the various analysis tools developed for rewrite logic specifications, while the translation into C allows for very efficient interpreters. A suite of tests show the performance of interpreters compiled from *K* definitions.

Key words: programming languages, rewriting logic, language interpreters.

1 Introduction

The K language definition framework [1] is a rewrite logic based framework for specifying programming languages. It includes both a notation, the K-notation, consisting of a series of domain-specific syntactic-sugar conventions aiming at simplifying and enhancing readability of language definitions, and a language definition technique, the K-technique, based on a first-order representation of continuations. As part of our ongoing research, we are developing a number of tools around K to assist in defining and analyzing programming languages.

⋆ Supported by NSF grants CCR 0234524, CCF-0448501, and CNS-0509321.
* Corresponding author.
Email addresses: mhills@cs.uiuc.edu (Mark Hills), tserban2@cs.uiuc.edu (Traian Şerbănuţă), grosu@cs.uiuc.edu (Grigore Roşu).

Here, we show two pieces of this work. First, we show the semantics of a simple programming language with functions defined using K. This language includes standard imperative features, including a controlled jump in the form of a function return. Second, we provide some details of a translation from our notation in K to an interpreter for the language, written in C. We are actively working on providing for the automated construction of interpreters from K definitions of languages, and currently have a semi-automated translation.

In Section 2, we present an overview of the K notation together with details of how it can be translated into rewrite logic. In Section 3 we show K at work by defining the Simple Imperative Language with Functions, or SILF. In Section 4 we provide details of our translation from K to C, including some initial performance figures of comparisons with equivalent programs written in other languages. Section 5 discusses future work and concludes the paper.

Related work. There are a number of different methods for specifying the semantics of programming languages, including operational methods such as Plotkin's SOS [2], denotational methods such as those from Scott and Strachey [3], Mosses's action semantics [4] and MSOS [5], and Meseguer and Roşu's rewriting logic semantics [6], among many others. There is also significant work on *executable* definitions of language semantics. An interesting example is Centaur [7], which includes a Prolog engine for executing formal language specifications. Another is ASF+SDF [8], which also uses term rewriting to define programming languages, but our contextual, continuation-based methodology, involving explicit access to the control state, appears quite different. One appealing aspect of rewriting logic semantics is that precisely the same rewrite logic definition of a language gives both an algebraic denotational semantics (an initial model semantics) and an operational semantics (the initial model is executable). Of the above, our work is most similar to rewrite logic semantics; more precisely, our framework can be regarded as a domain-specific syntactically sugared rewriting logic semantical framework.

2 The K Language Definition Framework

Here we briefly recall the *K-framework* [1], useful to compactly, modularly and intuitively define languages in rewrite logic. It consists of the *K-notation*, i.e., a series of notational conventions for matching modulo axioms, for eliding unnecessary variables, for sort inference, and for *c(K)ontext transformers*, and of the *K-technique*, which is a *c(K)ontinuation-based* technique to define languages algebraically. The K-framework is described in detail in [1].

Matching Modulo. Despite its general intractability [9], matching modulo *Associativity*, *Commutativity*, and *Identity*, or *ACI-matching*, tends to be relatively efficient in practice. Many rewrite engines support it in its full generality. ACI-matching leads to compact and elegant, yet efficiently executable

specifications. Different languages have different ways to state that binary operations are associative and/or commutative and/or have identities; to keep the discussion generic, we assume that all ACI operations are written using the *mixfix* concatenation notation "$\_\ \_$" and have identity "$\cdot$", while all but one [1] of the AI operations use the comma notation "$\_\,,\_$" and have identity written also "$\cdot$". In particular implementations of K specifications, to avoid confusion one may want to use different names for the different ACI or AI operations. ACI operations correspond to multi-sets, while the AI operations correspond to lists. Therefore, for any sort *Sort*, we tacitly add supersorts "*SortSet*", "*SortNeSet*", "*SortList*", and "*SortNeList*" of *Sort* (with the "*Ne*" versions being *non-empty*), constant operations "$\cdot : \rightarrow SortSet$" and "$\cdot : \rightarrow SortList$", and ACI operation "$\_\ \_ : SortSet \times SortSet \rightarrow SortSet$" and AI operation "$\_\,,\_ : SortList \times SortList \rightarrow SortList$" both with identities "$\cdot$".

ACI operations will be used to define states as "soups" of attributes; e.g., the state of a language can be a "soup" containing a store, locks which are busy, input/output buffers, etc., as well as a set of threads. Soups can be nested; for example, a thread may contain itself a soup of thread attributes, such as an environment, a set of locks that it holds, several stacks (for functions, exceptions, loops, etc.); an environment is further a soup of pairs (name,location), etc. Lists will be used to specify structures where the order of the attributes matters, such as buffers (for input/output), parameters of functions, etc.

For example, let us define an operation $update : Environment \times Name \times Location \rightarrow Environment$, where *Environment* is the set sort *NameLocationSet* associated to a pairing sort *NameLocation* with one constructor pairing operation $(\_\,,\_) : Name \times Location \rightarrow NameLocation$. $update(Env, X, L)$ is the same as *Env* except in the location of X, which should be replaced by L:

$$(\forall X : Name;\ L, L' : Location;\ Env : Environment)$$
$$update((X, L')\ Env,\ X,\ L) = (X, L)\ Env.$$

The ACI-matching algorithm "knows" that the first argument of *update* has an ACI constructor, so it will be able to match the lhs of this equation even though the pair (X, L') does *not* appear on the first position in the environment.

Sort Inference. Surprisingly, the variable declarations part of the equation of *update* takes almost half the size of the equation. It is often the case in our experiments with defining languages in Maude that variable declarations take a significant amount of space, sometimes more than half the entire language specification. However, in most cases *the sorts of variables can be automatically inferred from the context*. To simplify this process, we assume that all variable names start with a capital letter. Consider, e.g., the two terms of the equation above, $update((X, L')\ Env,\ X,\ L)$ and $(X, L)\ Env$. Since the arity of *update*

[1] The exception to the comma notation for AI operations will be the "continuation"; defined later, it will follow, just for ease of reading, the notation $\_ \curvearrowright \_$.

is $\mathit{Environment} \times \mathit{Name} \times \mathit{Location} \to \mathit{Environment}$, one can immediately infer that the sorts of X and L are *Name* and *Location*, respectively. Further, since the first argument of *update* has the sort *Environment* and since environments are constructed using the operation $\_\ \_ : \mathit{Environment} \times \mathit{Environment} \to \mathit{Environment}$, one can infer that the sort of *Env* is *Environment*.

Because of subsorting, a variable occurring on a position in a term may have multiple sorts. For example, the variable *Env* above can have both the sort *Environment* (which aliases *NameLocationSet*) and the sort *NameLocation*. The report [1] discusses in more depth the subtleties of sort inference in the presence of subsorting. Here we only recall that if an occurrence of a variable can have multiple sorts, we assume by default, or by convention, that that variable occurrence has *the largest* sort among those that it can have; this convention corresponds to the intuition that we assume the "least" information about each variable occurrence. If the same variable appears on multiple positions then we infer for that variable the "most concrete" sort that it can have among them. Technically, this is the intersection of all the largest sorts inferred for that variable on the different positions where it appears. If the variable sort-inference process is ambiguous, or if one is not sure, or if one really wants a different sort than the inferred one, or even simply for clarity, one is given the possibility to sort variables "on-the-fly": we append the sort to the variable using ":", e.g., $X : \mathit{Sort}$. For example, from the term $\mathit{update}(\mathit{Env}, X, L)$ one can only infer that the sort of *Env* is *Environment*, the most general possible under the circumstances. If for any reason one wants to refer to a "special" environment of just one pair, then one can write $\mathit{update}(\mathit{Env} : \mathit{NameLocation}, X, L)$.

Underscore Variables and Tuples. With the sort inference conventions, the equation defining the operation *update* can be therefore written as

$$\mathit{update}((X, L')\ \mathit{Env},\ X,\ L) = (X, L)\ \mathit{Env}.$$

Note that the location L' that occurs in the lhs is not needed; it is only used for "structural" purposes, i.e., it is there only to say that the name X is allocated at some location, but we do not care what that location is (we change it anyway). Since this will be a common phenomenon in our language definitions, we take the liberty to replace unnecessary letter variables by underscores, like in Prolog. Therefore, the equation above can be written

$$\mathit{update}((X, \_)\ \mathit{Env},\ X,\ L) = (X, L)\ \mathit{Env}.$$

Like we need to pair names and locations to create environments, we will often need to tuple two or more terms in order to "save" current information for later processing. In K, by convention we allow all tupling operations without defining them explicitly. Like the sorts of variables, their arities can also be inferred from the context. Concretely, if the term $(X_1 : \mathit{Sort1},\ X_2 : \mathit{Sort2},\ \ldots,\ X_n : \mathit{Sortn})$ appears in some context (the variable sorts may be in-

ferred), then we implicitly add to the signature the sort *Sort1Sort2...Sortn* and the operation $(\_,\_,\ldots,\_) : \mathit{Sort1} \times \mathit{Sort2} \times \cdots \times \mathit{Sortn} \rightarrow \mathit{Sort1Sort2...Sortn}$.

Contextual Notation for Rewrite Rules. All the subsequent rewrite rules will apply on just one (large) term, encoding the state of the program. Specifically, most of them will apply on subterms selected via matching, but only if the structure of the state permits it. In other words, most of our rules will be of the form $C[t_1]\cdots[t_n] \rightarrow C[t'_1]\cdots[t'_n]$, where C is some context term with $n \geq 0$ "holes" and t_1, ..., t_n are subterms that need to be replaced by t'_1, ..., t'_n in that context. C needs not match the entire state, but nevertheless sometimes it can be quite large. To simplify notation and ease reading, in K we write rules as $C[\frac{t_1}{t'_1}]\cdots[\frac{t_n}{t'_n}]$. This notation follows a natural intuition: first write the state context in which the transformation is intended to take place, then underline what needs to change, then write the changes under the line. Our contextual notation above proves to be particularly useful when combined with the "_" variables: if "_" appears in a context C, then it means that we do not care what is there but that we do not change it either.

Matching Prefixes, Suffixes and Fragments. We here introduce one more piece of notation that will help us further compact our language definitions by eliminating the need to mention unnecessary underscore variables. Many state attribute "soups" will be wrapped with specific operators to keep them distinct from other soups. For example, environments will be wrapped with an operation $\mathit{env} : \mathit{Environment} \rightarrow \mathit{Attribute}$ before they are placed in their threads' state attribute soup. Thus, if we want to find the location of a name X in the environment, then we match the environment attribute against the "pattern" term $\mathit{env}((X, L)\ \_)$ and thus find the desired location L; the underscore variable matches the rest of the environment. The underscores make pattern terms look heavier and harder to read than needed, especially when the state is defined using deeply nested soups of attributes (not the case in this paper). What one really wants to say above is that one is interested in the pair (X, L) that appears somewhere in the environment. In our particular domain of language definitions, we believe, subjectively, that the notation $\mathit{env}\langle(X, L)\rangle$ for the same pattern term is better than the one using the underscores. By convention, whenever "$\_ \circ \_$" is an ACI or AI operator wrapped by some attribute operator, say *att*, we write

$\mathit{att}(T\rangle$ (i.e., left parenthesis right angle) as syntactic sugar for $\mathit{att}(T \circ \_)$,
$\mathit{att}\langle T)$ (i.e., left angle right parenthesis) as syntactic sugar for $\mathit{att}(\_ \circ T)$,
$\mathit{att}\langle T\rangle$ (i.e., left and right angles) as syntactic sugar for $\mathit{att}(\_ \circ T \circ \_)$.

If "$\_\circ\_$" is an ACI operator then the three notations above have the same effect, namely that of matching T inside the soup wrapped by *att*; for simplicity, in

this case we just use the third notation, $att\langle T\rangle$. The intuition for this notation comes from the fact that the left and the right angles can be regarded as some hybrid between corresponding "directions" and parentheses. For example, if "$\_ \circ \_$" is AI (not C) then $(T\rangle$ can be thought of as a list starting with T (the left parenthesis) and continuing however it wishes (the right angle); in other words, it says that T is the *prefix* of the list wrapped by the attribute *att*. Similarly, $\langle T)$ says that T is a *suffix* and $\langle T\rangle$ says that T is a contiguous *fragment* within the list wrapped by *att*. If "$\_ \circ \_$" is also commutative, i.e., an ACI operator, then the notions of prefix, suffix and fragment are equivalent, all saying that T is a subset of the set wrapped by *att*.

This notational convention will be particularly useful in combination with other conventions part of the K notation. For example, the input and output of the programming language defined in the sequel will be modeled as comma separated lists of integers, using an AI binary operation "$\_ , \_$" of identity "$\cdot$"; then in order to read (consume) the next two integers N_1, N_2 from the input buffer, or to output (produce) integers N_1, N_2 to the output buffer, all one needs to do (as part of a larger context that we do not mention here) is:

$$in(\underset{\cdot}{N_1, N_2}\rangle \qquad \text{and, respectively,} \qquad out\langle\underset{N_1, N_2}{\cdot}) $$

The first matches the first two integers in the buffer and removes them (the "$\cdot$" underneath the line), while the second matches the end of the buffer (the "$\cdot$" above the line) and appends the two integers there. Note that the later works because of the matching modulo identity: $out\langle\cdot)$ is a shorthand for $out(\_ , \cdot)$, where the underscore matches the entire list; replacing "$\cdot$" by the list N_1, N_2 is nothing but appending the two integers to the end of the list wrapped by *out*. As another interesting example, this time using an ACI operator, consider changing the location of an identifier I in the environment to another location, say L; this could be necessary in the definition of a language allowing declarations of local variables, when a variable with the same identifier, I, is declared locally and thus "shadows" a previously declared variable with the same name. This can be done as follows (part of a larger context): $env\langle(I, \underset{L}{\_})\rangle$.

Context Transformers are the most subtle aspect of the K notation, based on the observation that, in programming language definitions, it is always the case that the state of the program does not change its significant structure during the execution of the program. For example, the store will always stay at the same level in the state structure, typically at the top level. If certain state infrastructure is known to stay unchanged during the evaluation of any program, and if one is interested in certain attributes that can be unambiguously located in that state infrastructure, then we only mention those attributes as part of the context assuming that the remaining part of the context can be generated automatically (statically). Since SILF does not have

Integer Numbers	$N ::=$	$(+\|\text{-})?(0..9)^+$
Declarations	$D ::=$	var I | var $I[N]$
Expressions	$E ::=$	$N \mid E + E \mid E$ - $E \mid E$ * $E \mid E$ / $E \mid E$ % $E \mid$ - $E \mid$ $E < E \mid E <= E \mid E > E \mid E >= E \mid E = E \mid E$!= $E \mid$ E and $E \mid E$ or $E \mid$ not $E \mid N \mid I(El) \mid I[E] \mid I \mid$ read
Expression Lists	$El ::=$	$E\ (,E)^* \mid nil$
Statements	$S ::=$	$I := E \mid I[E] := E \mid$ if E then S fi | if E then S else S fi | for $I := E$ to E do S od | while E do S od | S; $S \mid D \mid$ $I(El) \mid$ return $E \mid$ write E
Function Declarations	$FD ::=$	function $I(Il)$ begin S end
Identifiers	$I ::=$	$(\text{a} - \text{zA} - \text{Z})(\text{a} - \text{zA} - \text{Z0} - 9)^*$
Identifier Lists	$Il ::=$	$I\ (,I)^* \mid void$
Programs	$Pgm ::=$	$S?\ FD^+$

Fig. 1. Syntax for SILF

threads, exceptions or other complex control sensitive language features, context transformers do not make a difference in this paper, so we do not discuss them in more detail. The reader interested in the role of context transformers in compactness and modularity of language definitions is referred to [1].

3 SILF: A Simple Imperative Language with Functions

Using the K notation, we now define a simple imperative language with functions, which we will herein refer to as SILF. The BNF syntax for SILF is shown in Figure 1. Note that a program is made up of an optional statement, which is assumed to be global variable declarations (not just any arbitrary statement), followed by one or more functions, one of which should be called `main`. We assume below that programs are well formed and type correct, and that we do not need to worry about issues such as precedence. We adopt the *mix-fix* notation for syntax in algebraic notation, with the standard conversion, adding a new sort for each non-terminal, and a new operation for each production. For instance, the declaration of a function will be: function _(_) begin _ end : $Id \times IdList \times Stmt \longrightarrow FunDecl$.

State Infrastructure. Since the rules in the semantics given below act on the SILF state, it is important to understand the state structure. The state of the program is made up of a number of "ingredients" in the state "soup", in this case all at the top level. The continuation, indicated by k, keeps track of the current control context. The *fstack* is the function stack, and holds information about the computation to resume on return – this is similar to a stack frame. The *env* and *genv* hold name to location

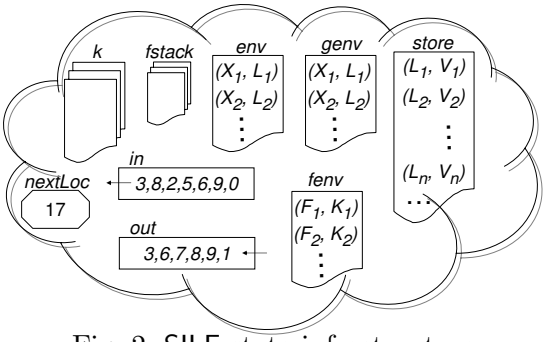

Fig. 2. SILF state infrastructure

mappings for the local and global environment, while the *fenv* holds mappings from function names to continuations for the bodies. The *store* holds location to value mappings. Input and output are represented by *in* and *out*, respectively. Finally, the next location in the store to allocate is tracked with *nextLoc*. This is represented graphically in Figure 2.

Formally, one declares the state structure by means of an algebraic signature, where each "ingredient" is wrapped by an appropriate operation that we call "attribute", and where ingredients are in the "soup" via an AC concatenation operation. Some of the soup ingredients are lists (e.g., I/O "buffers", function stacks, continuations), others are sets (e.g., environments, stores), while others are just plain numbers (e.g., the next location). Like the mix-fix algebraic signature associated to the BNF in Figure 1, we do not define the state signature here either, because it is straightforward.

When a program is executed, we need to construct its initial state. We do this using an *eval* operation. For SILF, this operation would take a program, *Pgm*, and an input list of integers, *Nl*, and "insert" them into a starting state:

$$\frac{\mathit{eval}(\mathit{Pgm}, \mathit{Nl})}{k(\mathit{Pgm})\ \mathit{fstack}(\cdot)\ \mathit{env}(\cdot)\ \mathit{genv}(\cdot)\ \mathit{fenv}(\cdot)\ \mathit{input}(\mathit{Nl})\ \mathit{output}(\cdot)\ \mathit{store}(\cdot)\ \mathit{nextLoc}(0)}$$

The continuation structure wrapped by k keeps an ordered list of tasks to be performed to continue the computation. We add additional sorts to represent the abstract syntax, including values (V), environments (Env), continuations (K), locations (L), and stores (Mem), with appropriate lists and sets for each.

Programs. A program is made up of a number of global variable declarations, followed by a number of functions. There is no inherent order to the functions – all functions can see all other functions. To execute a program, we need to process all global variable declarations, create the global environment, process all function declarations, and then invoke the main function:

$$k\left(\frac{\mathit{pgm}(S\ \mathit{FDs})}{\mathit{stmt}(S) \curvearrowright \mathit{mkGenv} \curvearrowright \mathit{fdecl}(\mathit{FDs}) \curvearrowright \mathit{stmt}(\mathsf{main}())}\right)$$

How $\mathit{stmt}(S)$ is processed is described later in this section. One can view *stmt* and *exp* as "compiling" the statement or expression, turning it into a continuation. As seen shortly, when S contains only variable declarations, $\mathit{stmt}(S)$ at the top of the continuation eventually produces a corresponding environment in the attribute *env*. Then, *mkGenv* only needs to move that environment into *genv*(this will allow us to easily refer to the global variable environment later):

$$k\left(\frac{\mathit{mkGenv}\rangle}{\cdot}\right)\ \mathit{env}(\mathit{Env})\ \mathit{genv}\left(\frac{\_}{\mathit{Env}}\right)$$

Function declarations are processed one by one:

$$\frac{\mathit{fdecl}(\mathit{FD{:}FunDecl}\ \mathit{FDs{:}FunDeclNeSet})}{\mathit{fdecl}(\mathit{FD}) \curvearrowright \mathit{fdecl}(\mathit{FDs})}$$

Functions. Function semantics cover three main constructs: function declaration, function invocation, and function return. We cover each below in turn. We first need to add the declared functions into the function environment. We do assume that function names are distinct and that declarations all occur at the start of the function. We add the necessary structure to the function body to bind the input values to the formal parameters, so we do not need to add this in the invocation semantics (the semantics of *bind* will be given shortly):

$$k(\ \frac{\mathit{fdecl}(\mathsf{function}\ I(\mathit{Is})\ \mathsf{begin}\ S\ \mathsf{end})}{\cdot}\ \rangle\ \mathit{fenv}\langle\ \frac{\cdot}{(I, \mathit{bind}(\mathit{Is}) \curvearrowright \mathit{stmt}(S))}\ \rangle$$

Functions can be used as either expressions or statements:

$$\frac{\mathit{exp}(I(\mathit{El}))}{\mathit{exp}(\mathit{El}) \curvearrowright \mathit{apply}(I)} \quad\Big|\quad \frac{\mathit{stmt}(I(\mathit{El}))}{\mathit{exp}(\mathit{El}) \curvearrowright \mathit{apply}(I) \curvearrowright \mathit{discard}}$$

The continuation item $\mathit{exp}(\mathit{El})$, when at the top of the continuation, evaluates the list of expressions *El* sequentially and produces their corresponding values, a term of the form $\mathit{val}(\mathit{Vl})$. When used as a statement, we put a discard continuation item into the continuation to throw away the return value (this will be defined shortly). Once the arguments have been evaluated, we can apply the function. Since functions are stored just as identifier/continuation pairs, we can just grab out the continuation for the function. Also, we save the current continuation and environment so we can quickly recover these when we exit the function on a return:

$$k(\mathit{val}(\_) \curvearrowright \frac{\mathit{apply}(I) \curvearrowright K}{K'})\ \mathit{fstack}(\frac{\cdot}{(\mathit{Env}, K)}\ \rangle\ \mathit{env}(\frac{\mathit{Env}}{\mathit{GEnv}})\ \mathit{fenv}\langle(I, K')\rangle\ \mathit{genv}(\mathit{GEnv})$$

When we encounter a return, first we need to evaluate the expression whose value we are returning. Once the value has been calculated, we can then switch context back to the caller, which we do by replacing the current environment and continuation with those saved at the top of the function stack:

$$\frac{\mathit{stmt}(\mathsf{return}\ E)\ )}{\mathit{exp}(E) \curvearrowright \mathit{return}} \quad\Big|\quad k(\mathit{val}(\_) \curvearrowright \frac{\mathit{return} \curvearrowright \_}{K})\ \mathit{fstack}(\frac{(\mathit{Env}, K)}{\cdot}\ \rangle\ \mathit{env}(\frac{\_}{\mathit{Env}})$$

State Helper Operations. Many of the rules in the SILF semantics perform similar changes to the state. We have abstracted these changes into a number of rules which can then be used across different parts of the semantics. The operation *bind* creates new bindings in the environment. This operation binds

a list of values to a list of identifiers, adding the identifier to the environment and the value to the store, linked by a shared location. To create a new binding in the environment without a value, we use a variant of the *bind* operation, which binds a list of identifiers to a list of locations but does not alter the store ($|\_|$ is the usual length operation on lists and Ll is the location list $(L, L+1, \ldots, L+|Il|-1)$):

$$k(\underline{\ val(Vl) \curvearrowright bind(Il)\ \rangle}_{\cdot}\ env(\underline{\ Env\ }_{Env[Il \leftarrow Ll]})\ store(\underline{\ Mem\ }_{Mem[Ll \leftarrow Vl]})\ nextLoc(\underline{\ L\ }_{L+|Il|})$$

$$k(\underline{\ bind(Is)\ \rangle}_{\cdot}\ env(\underline{\ Env\ }_{Env[Is \leftarrow locs(L, len(Is))]})\ nextLoc(\underline{\ L\ }_{L+len(Is)})$$

The $[\_ \leftarrow \_]$ operation will properly update the set, using the list on the left as a list of "keys" to either add a new key/value pair to the set or replace an existing key/value pair with a new pair. The definition is straightforward, and is not shown here.

We can also bind blocks of storage. This will just bind the first location to the identifier and then advance the next location an arbitrary amount. This can be used to represent allocating a block of memory for an array.

$$k(\underline{\ val(int(N)) \curvearrowright bindBlock(I)\ \rangle}_{\cdot}\ env(\underline{\ Env\ }_{Env[I \leftarrow L]})\ nextLoc(\underline{\ L\ }_{L+N})$$

For assignment, *assignTo* assigns a value to the store in two steps, first converting identifier assignment(*assignTo*) to location assignment (*assignToLoc*) then carrying out the assignment:

$$k(val(V) \curvearrowright \underline{\ assignTo(I)\ \rangle}_{assignToLoc(L)}\ env\langle(I, L)\rangle$$

$$k(\underline{\ val(V) \curvearrowright assignToLoc(L)\ \rangle}_{\cdot}\ store(\underline{\ Mem\ }_{Mem[L \leftarrow V]})$$

We also have a similar version for arrays, which will assign at an offset.

$$k(\underline{\ val(int(N), V) \curvearrowright arrayAssign(I)\ \rangle}_{val(V) \curvearrowright assignToLoc(L+N)}\ env\langle(I, L)\rangle$$

Similarly we have two lookup operations:

$$k(\underline{\ lookupLoc(L)\ \rangle}_{val(V)}\ store\langle(L, V)\rangle \qquad k(\underline{\ val(int(N)) \curvearrowright lookupOffset(I)\ \rangle}_{lookupLoc(L+N)}\ env\langle(I, L)\rangle$$

Occasionally we will want to discard a value from the continuation. To do so, we use *discard* with the following semantics: $k(\underline{\ val(V) \curvearrowright discard\ }\rangle$
.

Variable Declarations. In SILF we have two different types of variable declarations – integers and integer arrays. Arrays can only be declared of a fixed (positive integer) size. In both cases, the declaration does *not* set an initial value – this corresponds to a concept of "junk" in the memory before assignment, and any read attempts of "junk" will fail. We treat arrays identically to C (arrays are 0 indexed, so an array of 10 elements is indexed from 0 to 9) with the location of the array name the same as location 0:

$$\frac{stmt(\mathsf{var}\ I)}{bind(I)} \qquad \left| \qquad \frac{stmt(\mathsf{var}\ I[N])}{val(int(N)) \curvearrowright bindBlock(I)} \right.$$

Lookups and Simple Expressions. Some of SILF's most basic expressions are lookups of name and indexed array values, as well as literal expressions. For a literal integer, we just return a value with the integer encapsulated in a value wrapper: $\dfrac{exp(N)}{val(int(N))}$. For both identifiers and arrays, we return the current value, either assigned to the identifier or to the given element of the array. We will process this in two steps, first retrieving the value's location, then retrieving the value:

$$k(\underline{\ exp(I)\ }\rangle\ env\langle(I, L)\rangle \qquad \left| \qquad \frac{exp(I[E])}{exp(E) \curvearrowright lookupOffset(I)} \right.$$
$$lookupLoc(L)$$

Arithmetic, Relational, and Logical Operations. All three operation types follow the same general pattern. When we encounter an addition expression, e.g., we first need to evaluate both operands. We also need to keep track of what operation we are performing. So, we will replace an expression such as $E + E'$ with one were we evaluate E and E' and put $+$ on the continuation to remind ourselves what we need to do with the results. Once we get back the values from evaluating the two expressions (here, expected to both be integers) on top of a $+$, we return their sum (using integer addition):

$$\frac{exp(E + E')}{exp(E, E') \curvearrowright +} \qquad \left| \qquad \frac{val(int(N), int(N')) \curvearrowright +}{val(int(N +_{int} N'))} \right.$$

Relational operators work identically to arithmetic operators, except we apply relational operations on the results and return boolean values:

$$\frac{exp(E < E')}{exp(E, E') \curvearrowright <} \qquad \left| \qquad \frac{val(int(N), int(N')) \curvearrowright <}{val(bool(N <_{int} N'))} \right.$$

Logical operations are handled almost exactly the same:

$$\frac{exp(E \text{ and } E')}{exp(E, E') \curvearrowright and} \qquad \frac{val(bool(B), bool(B')) \curvearrowright and}{val(bool(B \ and_{bool} \ B'))}$$

All the arithmetic, relational, and logical operations are defined in Appendix A.

Assignment Statements. SILF has two types of assignment:

$$\frac{stmt(I := E)}{exp(E) \curvearrowright assignTo(I)} \qquad \frac{stmt(I[E] := E')}{exp(E, E') \curvearrowright arrayAssign(I)}$$

Conditional Statements. SILF has two conditionals, one with just a true branch, one with true and false branches. We convert the first into the second:

$$\frac{\text{if } E \text{ then } St \text{ fi}}{\text{if } E \text{ then } St \text{ else skip fi}} \quad \text{where skip has the usual semantics:} \quad k(\frac{stmt(\text{skip})}{\cdot})\rangle$$

For the general conditional, we first evaluate the condition, "compiling" the two branches and storing them in the continuation, wrapped by *if(_,_)*:

$$\frac{stmt(\text{if } E \text{ then } St \text{ else } Sf \text{ fi})}{exp(E) \curvearrowright if(stmt(St), stmt(Sf))}$$

If the result is true, then we will evaluate the first branch (which we have already converted into a continuation), and if false we will evaluate the second:

$$\frac{val(bool(\text{true})) \curvearrowright if(Kt, Kf)}{Kt} \qquad \frac{val(bool(\text{false})) \curvearrowright if(Kt, Kf)}{Kf}$$

Loop Statements. We transform "for" loops into "while" loops:

$$\frac{\text{for } I := E_1 \text{ to } E_2 \text{ do } S \text{ od}}{I := E_1; \text{while } I \leq E_2 \text{ do } S \ ; \ I :=I + 1 \text{ od}}$$

We give semantics to "while" loops by changing the while statement into a while continuation that contains the ("compiled") guard expression and the while body, at the same time evaluating the guard:

$$\frac{stmt(\text{while } E \text{ do } S \text{ od})}{exp(E) \curvearrowright while(exp(E), stmt(S))}$$

Next, based on whether the guard evaluates to true or false, we do or do not need to evaluate the body of the while:

$$\frac{val(bool(\text{true})) \curvearrowright while(Ke, Ks)}{Ks \curvearrowright Ke \curvearrowright while(Ke, Ks)} \qquad \frac{val(bool(\text{false})) \curvearrowright while(Ke, Ks)}{\cdot}$$

I/O Statements. SILF allows for rudimentary I/O, with the ability to read and write integers. For input, we take the next available integer:

$$k(\ \underline{exp(\mathsf{read})}\ \rangle\ input(\ \underline{N}\ \rangle \quad \begin{array}{c} \underline{\quad} \\ val(int(N)) \end{array} \qquad \begin{array}{c} \underline{\quad} \\ \cdot \end{array}$$

For output, we evaluate the expression, then add it to the *end* of the output:

$$\frac{stmt(\mathsf{write}\ E)}{exp(E) \curvearrowright write} \quad \Big| \quad k(\ \underline{val(int(N)) \curvearrowright write}\ \rangle\ output\langle\ \underline{\cdot}\ ) \quad \begin{array}{c} \underline{\quad} \\ \cdot \end{array} \qquad \begin{array}{c} \underline{\quad} \\ N \end{array}$$

Sequential Composition is straightforward: $\dfrac{stmt(S; S')}{stmt(S) \curvearrowright stmt(S')}$

4 Towards Automatic Synthesis of Language Interpreters

An important goal which we set for the K framework is that it should allow us to automatically generate efficient interpreters from language definitions. While this goal is still ahead of us, here we briefly present the semi-automatic generation of an interpretor for SILF.

Preprocessing. We currently assume as input a well-formed, type-checked program, which is then preprocessed to yield a simpler yet semantically equivalent program. During preprocessing, identifiers are replaced by numbers and variable declarations by memory allocation commands. Integers are wrapped (e.g., $\mathsf{i}(0)$ for 0), and functions are named with indices and parameter list sizes to aid with allocation (e.g., $\mathsf{f}(3)(5)$ for function number 3 with 5 parameters). This essentially eliminates the environment, which is now just an index into the store, similar to a frame pointer. We can best illustrate this with an example. In Figure 3, we have a program in SILF. In Figure 4, we have the equivalent program after translation. Note that translation can be performed statically and automatically.

Precompilation and instruction generation. We chose to clearly divide the semantic rules into *precompilation* and *execution* rules. The precompilation phase reduces the program to a continuation, which the execution phase then runs to modify the state. In our case, we can divide the semantic rules into two groups: those in which the left-hand-side is a state and those in which it is a continuation. We precompile only the latter, dividing each language task (e.g., assignment, function call) into a series of smaller tasks. Bytecode is then generated from a precompiled form of the program by a process of flattening, translating the graph-like structure of the continuation into an array. The bytecode "instructions" are given by the continuation items. This process is mostly automatic, with our instructions determining the structure of the virtual machine.

```
function writeBinary(x) begin
  var i;
  var b[32];
  var j;
  i := 0;
  while x > 0 do
    b[i] := x % 2;
    x := x / 2;
    i := i + 1
  od
  j := i - 1;
  while j >= 0 do
    write b[j];
    j := j - 1
  od
end
function main(void) begin
  writeBinary(read)
end
```

Fig. 3. Source Language Program

```
globals(0) ;
function f(1)(1) {
  alloc(1) ;
  alloc(32) ;
  alloc(1) ;
  l(i(1)) := i(0) ;
  while l(i(0)) > i(0) do {
    l(l(i(1)) + i(1)) := l(i(0)) % i(2) ;
    l(i(0)) := l(i(0)) / i(2) ;
    l(i(1)) := l(i(1)) + i(1)
  } ;
 l(i(34)) := l(i(1)) - i(1) ;
  while l(i(34)) >= i(0) do {
   writeInt(l(l(i(34)) + i(1)));
   l(i(34)) := l(i(34)) - i(1)
  }
}
function f(0)(0) {
  f(1)(readInt)
}
```

Fig. 4. Translated Program

Execution. The execution rules act on a modified version of the state, with a separate stack for values and a control stack for continuations. This requires a change in some of the rules, which we believe can be automated. This then aligns with the interpreted view of the rules, with stores and stacks represented as arrays, and stack operations represented as array index manipulation. The interpreter executes program by referencing the item on top of the continuation and the values on top of the stacks, which uniquely determine the rule to apply (with the continuation item alone determining most of the rules). The virtual machine then executes an infinite loop which selects the next continuation item and runs the code for the selected rule.

Evaluation. For evaluation we have chosen several programs, each exercising different execution tasks. *perm* is an all-permutations generation algorithm using recursive backtracking with globals and returns. *binary* computes the base two representation for all numbers up to the input number by successive divisions by 2, and exercises iterative function calls with local array declarations. *sieve* is the Eratosthenes' sieve algorithm for computing primes up to the input

Program	K to Maude	K to C	BC	C	Java
perm(6)	80.840	0.048	0.155	0.003	0.174
perm(9)	*	45.560	154.016	1.615	11.342
binary(1,000)	17.037	0.019	0.100	0.004	0.190
binary(1,000,000)	*	32.631	209.949	4.955	55.782
sieve(10,000,000)	*	27.671	-	1.199	3.591
hanoi(23)	*	18.140	86.432	4.394	57.761

Execution times in seconds. − indicates test not performed, ∗ indicates test timed out. Evaluation performed on Intel® Pentium® 4 CPU 2.00GHz with 1GB RAM, gcc version 3.3.6, compilation flags: -O3 -march=pentium4 -pipe -fomit-frame-pointer

Fig. 5. Evaluation Results

number, which exercises addressing large arrays. Finally, *hanoi* is the standard recursive solution for the Hanoi towers problem, exercising recursive functions. Results are shown in Figure 5. We don't have results for *BC* on *sieve*, since *BC* only allows 16 bit array indexes. The C interpreter for SILF outperforms *BC* and is competitive with *C*, and occasionally outperforms *Java* (additional work is needed to determine under what circumstances). *Maude*'s times are higher because of extensive *ACI*-matching, reducing speeds from millions of rewrites to around tens of thousands of rewrites per second. Because of this, we do not have figures for Maude for the larger test cases.

5 Conclusions

In this paper we introduced the K language definition framework and used it to define a simple imperative language with functions. We also showed an example of translating this definition into an interpreter in C. Based on current encouraging results, we believe this is a promising strategy for automatically deriving interpreters from definitions of language semantics.

References

[1] G. Roşu, K: a Rewrite Logic Framework for Language Design, Semantics, Analysis and Implementation, Tech. Rep. UIUCDCS-R-2005-2672, Computer Science Dept., Univ. of Illinois at Urbana-Champaign (2005).

[2] G. D. Plotkin, A structural approach to operational semantics, Journal of Logic and Algebraic Programming 60-61 (2004) 17–139.

[3] J. E. Stoy, Denotational semantics: the Scott-Strachey approach to programming language theory, MIT Press, 1977.

[4] P. D. Mosses, Action Semantics, no. 26 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1992.

[5] P. D. Mosses, Modular structural operational semantics, Journal of Logic and Algebraic Programming 60-61 (2004) 195–228.

[6] J. Meseguer, G. Roşu, Rewriting Logic Semantics: From Language Specifications to Formal Analysis Tools , in: IJCAR'04, Springer LNAI 3097, 2004, pp. 1–44.

[7] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, V. Pascual, Centaur: the system, in: SDE 3: Proceedings of the third ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments, ACM Press, New York, NY, USA, 1988, pp. 14–24.

[8] M. G. J. van den Brand, J. Heering, P. Klint, P. A. Olivier, Compiling language definitions: the ASF+SDF compiler, ACM Trans. Program. Lang. Syst. 24 (4) (2002) 334–368.

[9] D. Kapur, P. Narendran, NP-Completeness of the Set Unification and Matching Problems., in: CADE'86, 1986, pp. 489–495.

A Additional Semantics Rules

Here we include the additional rules in the semantics for SILF which were not included above. These rules are similar to those shown in Section 3.

A.1 Arithmetic Operations

$$\frac{exp(E + E')}{exp(E, E') \curvearrowright +}$$

$$\frac{val(int(N), int(N')) \curvearrowright +}{val(int(N +_{int} N'))}$$

$$\frac{exp(E - E')}{exp(E, E') \curvearrowright -}$$

$$\frac{val(int(N), int(N')) \curvearrowright -}{val(int(N -_{int} N'))}$$

$$\frac{exp(E * E')}{exp(E, E') \curvearrowright *}$$

$$\frac{val(int(N), int(N')) \curvearrowright *}{val(int(N *_{int} N'))}$$

$$\frac{exp(E/E')}{exp(E, E') \curvearrowright /}$$

$$\frac{val(int(N), int(N')) \curvearrowright /}{val(int(N/_{int}N'))}$$

$$\frac{exp(E\%E')}{exp(E, E') \curvearrowright \%}$$

$$\frac{val(int(N), int(N')) \curvearrowright \%}{val(int(N\%_{int}N'))}$$

$$\frac{exp(-E)}{exp(E) \curvearrowright u-}$$

$$\frac{val(int(N)) \curvearrowright u-}{val(int(-_{int}N))}$$

A.2 Relational Operations

$$\frac{exp(E < E')}{exp(E, E') \curvearrowright <}$$

$$\frac{val(int(N), int(N')) \curvearrowright <}{val(bool(N <_{int} N'))}$$

$$\frac{exp(E <= E')}{exp(E, E') \curvearrowright <=}$$

$$\frac{val(int(N), int(N')) \curvearrowright <=}{val(bool(N <=_{int} N'))}$$

$$\frac{exp(E > E')}{exp(E, E') \curvearrowright >}$$

$$\frac{val(int(N), int(N')) \curvearrowright >}{val(bool(N >_{int} N'))}$$

$$\frac{exp(E >= E')}{exp(E, E') \curvearrowright >=}$$

$$\frac{val(int(N), int(N')) \curvearrowright >=}{val(bool(N >=_{int} N'))}$$

$$\frac{exp(E = E')}{exp(E, E') \curvearrowright =}$$

$$\frac{val(int(N), int(N')) \curvearrowright =}{val(bool(N =_{int} N'))}$$

$$\frac{exp(E != E')}{exp(E, E') \curvearrowright != }$$

$$\frac{val(int(N), int(N')) \curvearrowright != }{val(bool(N !=_{int} N'))}$$

A.3 Logical Operations

Note that these operations are *not* short-circuit, since we evaluate both operands to *and* and *or* at once. We could make them short-circuit by instead evaluating only the first operand, and storing the second with the continuation for the operator. Based on the result of evaluating the first operand, we could then either return the proper value or evaluate the second operand to give us the value of the operation.

$$\frac{exp(E \text{ and } E')}{exp(E, E') \curvearrowright and}$$

$$\frac{val(bool(B), bool(B')) \curvearrowright and}{val(bool(B \; and_{bool} \; B'))}$$

$$\frac{exp(E \text{ or } E')}{exp(E, E') \curvearrowright or}$$

$$\frac{val(bool(B), bool(B')) \curvearrowright or}{val(bool(B \ or_{bool} \ B'))}$$

$$\frac{exp(\text{not } E)}{exp(E) \curvearrowright not}$$

$$\frac{val(bool(B)) \curvearrowright not}{val(bool(not_{bool} \ B))}$$