14th International Congress on Cybernetics
Namur (Belgium), Aug.21-25, 1995, Preprint

Reduction of Language Hierarchies

Robert Glück
Department of Computer Science
University of Copenhagen
DK-2100 Copenhagen, Denmark
e-mail: glueck@diku.dk

Andrei Klimov[1]
Keldysh Institute of Applied Mathematics
Russian Academy of Sciences
RU-125047 Moscow, Russia
e-mail: And.Klimov@refal.ac.msk.su

Abstract We study the structure of language hierarchies and their reduction by two forms of metacomputation in order to overcome the time and space complexity of language hierarchies. We show that program specialization and program composition are sufficient to reduce all forms of language hierarchies constructed from interpreters and translators. We argue that the reduction of language hierarchies is a prerequisite for effective formal linguistic modeling on a large scale.

1 Introduction

One of the defining features of modern science is the use of languages, both informal and formal, to construct *linguistic models* of reality [8]. The introduction of the computer was a revolutionary step in the execution of formal linguistic models and, as a result, the number of linguistic models created and used has significantly increased in all branches of science during the last decades. Computer science, as we see it, is laying the foundations and developing the research paradigm and scientific method of formal linguistic modeling. Linguistic models that can be performed by a computer, at least in principle, are referred to as programs, or algorithms.

Languages and their definitions play a central role in all forms of linguistic modeling. A modern approach in computer science for solving wide-spectrum problems is to devise application-oriented languages that make it easy for the user to express computational tasks in a particular area. Such languages are often defined by using several interpreters, or by translating them to a ground language via a series of intermediate languages. But hierarchies of languages are not a simplification in terms of the underlying designation process, but increase computational complexity: a statement of a higher level language is usually defined by a sequence of actions (or phrases) on a lower, more elementary level. The effective and efficient reduction of language hierarchies is a prerequisite for formal linguistic modeling on a large scale.

This contribution addresses the problem of systematically reducing the computational costs of language hierarchies by two forms of metacomputation: *program composition* and *program specialization*. We shall not be concerned *how*, but *what* has to be achieved by metacomputation. The presented problems can be seen as 'test cases' for existing methods and as a guideline for further research. More specifically, we study the reduction of *homogeneous hierarchies*, i.e. hierarchies of translators or interpreters only, as well as *heterogeneous hierarchies* in which translative and interpretive definitions may occur in any order. For the sake of completeness, we also summarize two approaches for converting translators into interpreters and vice versa. From now on we shall refer to formal languages simply as languages.

This work belongs to a line of research which aims at a better understanding of metacomputation and the use of metasystem transition, e.g. [4-6, 8-11].

[1] Supported by the Russian Basic Research Foundation under grant number 93-01-628.

2 Hierarchical Systems of Languages

Data, programs and application We assume a fixed set D of *data* is given, which can represent programs written in different languages, as well as their input and output. We shall assume nothing further about data; we could chose symbol strings, Lisp lists, etc. To express the *application* of an L-program to its input we use angular brackets, e.g. `<Pgm Input>`$_{\mathbf{L}}$ = `Output`. We omit the language index L when it is not essential. Capitalized names in `typewriter` font denote elements of the data domain, e.g. `Pgm` $\in D$. Two expressions are considered equal if they reduce to identical elements of the data domain (or both sides are undefined).

Language definitions We know of exactly two forms of language definitions: interpretive and translative. An interpreter defines a source language A by actions in another language B, while a translator defines a source language A by translation to a target language B where the translation is described in a meta-language M.

Definition 1 (Interpretation). A B-program `Int` is an A/B-*interpreter* if for every A-program `P` $\in D$ and every input `X` $\in D$

`<P X>`$_{\mathbf{A}}$ = `<Int P, X>`$_{\mathbf{B}}$

Definition 2 (Translation). An M-program `Trans` is an A→B-*translator* if

1) `<Trans P>`$_{\mathbf{M}} \in$ B-programs for all A-programs `P` $\in D$, and
2) `<<Trans P>`$_{\mathbf{M}}$ `X>`$_{\mathbf{B}}$ = `<P X>`$_{\mathbf{A}}$ for all A-programs `P` $\in D$ and every input `X` $\in D$.

Language hierarchies A hierarchical system of languages is a series of consecutive definitions where each definition is either an interpreter, or a translator. The hierarchy starts with a language N on the highest level and ends at a ground language 0. This is illustrated below where Def_{I} is either an I/J-interpreter or an I→J-translator (written in some lower language). We require that each language hierarchy is 'closed' in the sense that the meta-languages of the translators can always be reduced to the ground language. For the sake of simplicity let all translators in a hierarchy be written in the same language M which is identical to the ground language 0.

$$\mathrm{N} \xrightarrow{Def_{\mathrm{N}}} \mathrm{N{-}1} \;\ldots\; \mathrm{I} \xrightarrow{Def_{\mathrm{I}}} \mathrm{J} \;\ldots\; 1 \xrightarrow{Def_{1}} 0$$

Correct hierarchies Language hierarchies, being sequences of interpreters and translators, obey certain typing rules. Let us denote by * a 'join' operation for building a language hierarchy. For example, Def_{A} * Def_{B} * Def_{C} is a three-level hierarchy constructed from three language definitions. The following rules hold for any pair of adjacent definitions in a correct hierarchy where _/_ is an interpreter, _→_ a translator, and A, B, C are languages.

(1) A/B * B/C
(2) A→B * B→C
(3) A/B * B→C
(4) A→B * B/C

Complexity of hierarchies Abstraction by means of language hierarchies is not a simplification in terms of time and space complexity, but on the contrary. A statement of a higher level language is usually defined by a sequence of actions (or phrases) on a lower, more elementary level. Generally speaking, the computational costs grow exponentially with the height of the interpreter hierarchy, and the size of a translated text grows exponentially with the height of the translator hierarchy. The ultimate goal is to minimize the complexity involved in language hierarchies. A hierarchy of formal language definitions is just a 'complicated program' composed from a series of interpreters and translators. Hence, methods of program transformation and optimization can be used to reduce its complexity.

3 Metacomputation

Metavariables To manipulate application expressions without specifying all data elements we introduce metavariables. A *metavariable* `m` ∈ *M* stands for an unspecified data element; it ranges over the whole domain *D*. Lowercase names in `typewriter` font denote elements of the metavariable domain, e.g. `m` ∈ *M*.

Metacoding In order to manipulate arbitrary expressions by programs we define an injective mapping, called *metacoding*, from program expressions (possibly including applications and metavariables) into the data domain. Metacoding plays the same role in metacomputation as Gödel numeration in logic. We shall use a two-dimensional notation by moving metacoded expressions down one line for each level of metacoding.

Metacomputation We refer to any process of simulating, analyzing and transforming programs by programs as *metacomputation*, a term that underlines the fact that this activity is one level higher than ordinary computation. Equivalence transformation of programs is the main possibility for metacomputation; in this paper we use *program composition* and *program specialization* (see e.g. [9,12,7,2]). We will not fix a particular method for metacomputation, but specify these two transformation tasks equationally.

Definition 3 (Program composition). An M-program `Cpo` is an A→B-*composer* if for every A-program `P`, `Q` ∈ *D*, every input `X` ∈ *D* and `x` ∈ *M*,

$$\langle \texttt{Cpo}\ \underset{\langle \texttt{P}\ \langle \texttt{Q}\ \texttt{x}\rangle\rangle_{\mathbf{A}}}{\_\_\_\_\_}\rangle_{\mathbf{M}} = \texttt{R} \qquad \textit{such that} \qquad \langle \texttt{R}\ \texttt{X}\rangle_{\mathbf{B}} = \langle \texttt{P}\ \langle \texttt{Q}\ \texttt{X}\rangle\rangle_{\mathbf{A}}$$

Definition 4 (Program specialization). An M-program `Spec` is an A→B-*specializer* if for every A-program `P` ∈ *D*, every input `X`, `Y` ∈ *D* and `y` ∈ *M*,

$$\langle \texttt{Spec}\ \underset{\langle \texttt{P}\ \texttt{X},\ \texttt{y}\rangle_{\mathbf{A}}}{\_\_\_\_\_}\rangle_{\mathbf{M}} = \texttt{R} \qquad \textit{such that} \qquad \langle \texttt{R}\ \texttt{Y}\rangle_{\mathbf{B}} = \langle \texttt{P}\ \texttt{X},\texttt{Y}\rangle_{\mathbf{A}}$$

Note that these definitions say nothing about the quality of the metacomputation process, but we expect that the transformations performed are more than trivial program composition or trivial program specialization since the success of reducing the complexity of language hierarchies depends on the power of the metacomputation methods. Formulas involving metacomputation are collectively referred to as *MST-formulas* (MST=metasystem transition [8]).

4 Converting Definitions to their Dual Form

Language definitions in mathematics usually take the translative form stated in some meta-language (an excellent example is [1]), while language definitions in computer science are often interpretive. Mathematics is an example of a linguistic activity where it is more convenient to define language extensions stepwise by translation into more elementary phrases than by defining the whole extended language interpretively. On the other hand, when defining a new language from scratch, as is often the case in computer science, it is usually easier to give its definition by an interpreter rather than by a translator. However, in both situations the need may arise for converting translators into interpreters, and vice versa. We summarize two approaches for converting language definitions into their dual form.

Converting translators to interpreters [6] Let `TransAB` be an A→B-translator written in C, and let B be defined by a B/C-interpreter `IntBC`. This is a two-level language hierarchy of the form `TransAB * TransBC`. The A→B-translator can be converted into an A/C-interpreter by program composition as follows. First, define an inefficient A/C-interpreter `IntAC` that performes A-programs in two stages: by translating an A-program into B and then by interpreting the B-program.

$$\textbf{def}\ \langle \texttt{IntAC}\ \texttt{p},\texttt{x}\rangle_{\mathbf{C}} = \langle \texttt{IntBC}\ \langle \texttt{TransAB}\ \texttt{p}\rangle,\texttt{x}\rangle_{\mathbf{C}}$$

Now let Cpo be a C→C-composer and p,x ∈ *M* be metavariables. By removing the intermediate language B by program composition a more efficient A/C-interpreter IntAC' may be obtained where <IntAC' P,X>$_{\mathbf{C}}$ = <IntAC P,X>$_{\mathbf{C}}$:

```
<Cpo _____________________> = IntAC'
     <IntBC <TransAB p>,x>C
```

Converting interpreters to translators [3] Let IntBC be an B/C-interpreter and let Spec be a C→C-specializer written in C. The B/C-interpreter can be converted into an B→C-translator by program specialization as follows. Let P be a B-program and define a C-program R such that <R X>$_{\mathbf{C}}$ = <P X>$_{\mathbf{B}}$ for all X ∈ *D*:

```
def <R x>C = <IntBC P,x>C
```

First MST An efficient C-program R' may be obtained by specializing R's definition:

```
<Spec _____________>C = R'
     <IntBC P,x>C
```

Second MST An B→C-translator TransBC, such that <<TransBC P> X>$_{\mathbf{C}}$ = <R' X>$_{\mathbf{C}}$, can be defined by replacing the metacoded program P by a metavariable p ∈ *M* in the first MST-formula:

```
def <TransBC p>C = <Spec _______p____>C
                        <IntBC •,x>C
```

An efficient B→C-translator TransBC' may be obtained by specializing TransBC's definition:

```
<Spec ____________________>C = TransBC'
     <Spec _______p____>C
          <IntBC •,x>C
```

5 Reducing Language Hierarchies

5.1 Homogeneous Hierarchies of Languages

We consider homogeneous language hierarchies and state their reduction for the two-level case. The same methods can be used for homogeneous hierarchies of arbitray height. Recall that in both cases the translator and interpreter resulting from the reduction of the hierarchy can be converted to its dual definition as explained in Section 4.

Translator hierarchies Let each language in a two-level hierarchy be defined by a translator. A text in the top language A is then translated by the translator TransAB into a text in the lower language B which is translated by TransBC into a text written in the ground language C. Thus, the language hierarchy has the form TransAB * TransBC.

An A→C-translator TransAC that translates A directly into C without the intermediate translation into B can be derived by metacomputation using program composition. Let Cpo be an M→M-composer and let p ∈ *M* be a metavariable. For simplicity we assume that both translators are defined in the same meta-language M.

```
<Cpo _____________________> = TransAC
     <TransBC <TransAB p>>M
```

Interpreter hierarchies Let each language in a two-level hierarchy be defined by an interpreter. A text in the top language A is interpreted by the interpreter IntAB written in the lower language B which in turn is interpreted by the interpreter IntBC described in the ground language C. Thus, the language hierarchy has the form IntAB * IntBC.

An A/C-interpreter IntAC that interprets A directly in C without the intermediate interpretation of B can be obtained by specialization of the interpreter IntBC with respect to IntAB. Let Spec be a C→C-specializer and let p, x ∈ *M* be metavariables that stand for an arbitray text and its input.

```
<Spec ________________________> = IntAC
      <IntBC IntAB, (p, x)>C
```

5.2 Heterogenous Hierarchies of Languages

We now consider the general case of heterogeneous hierarchies that consist of translative and interpretive definitions in an arbitrary order and show that all heterogeneous hierarchies can be reduced in three steps either to an interpreter or a translator. A heterogeneous hierarchy contains at least an interpreter and a translator.

(1) Interpreter over translators First, each subhierarchy of the form "interpreter over translator" is reduced to a single interpreter that replaces both definitions. Assume that the language A is interpreted by an A/B-interpreter IntAB which in turn is defined by a B→C-translator. Thus, the language hierarchy has the form IntAB * TransBC. An A/C-interpreter IntAC which interprets A directly in C can be obtained by *ordinary computation*: by translating the interpreter IntAB into C.

```
<TransBC IntAB> = IntAC
```

By repeatedly using this reduction, any subhierarchy of the form

```
... IntI_ *  Trans__  * ... *  Trans_J ...
   \____interpreter over translators____/
```

where I is the top language and J the ground language, can be reduced to a single I/J-interpreter IntIJ.

(2) Translators over interpreters It is easy to verify that the hierarchy resulting from step (1) consists of exactly two homogeneous subhierarchies: a sequence of translators followed by a sequence of interpreters.

```
TransN_ * ... * Trans_I * IntI_ * ... * Int_0
\____translative part____/    \__interpretive part__/
```

Further reduction requires metacomputation over each subhierarchy. The two homogeneous subhierarchies can be reduced to a translator (by a program composer) and to an interpreter (by a program specializer), respectively, using the methods explained in Section 5.1.

(3) Two-level hierarchy After step (2) the language hierarchy is reduced to a two-level hierarchy consisting of one translator TransNI and one interpreter IntI0.

```
 TransNI * IntI0
\___two levels___/
```

The final reduction to a single interpreter can be obtained using a 0→0-composer Cpo as shown in Section 4 (recall that we assumed that the meta-language of translators can be reduced to the ground language 0):

```
<Cpo ________________________> = IntN0'
     <IntI0 <TransNI p>, x>0
```

As a result, the heterogeneous hierarchy consisting of translative and interpretive definitions in an arbitrary order has been reduced to a single interpreter. If necessary, the interpreter IntN0 can be converted to an N→0-translator TransN0 (Section 4).

6 Summary

We summarize the reduction formulas stated in the previous sections. In each case the pair interpreter/translator can be folded into a single definition. The formulas 1 and 2 describe the two homogeneous cases, while the formulas 3 and 4 describe the two heterogeneous cases. The last case describes the conversion of interpreters into translators. We list the simpler case for each pair of language definitions; the reduced definition can be converted into its dual definition if needed. We use the following shorthand notation for A/B-interpreters and A→B-translators written in M:

interpreter: A/B translator: A→B (M)

	Type of reduction	*MST-formula*	*Note*
1	A→B * B→C ⇒ A→C (M) (M) (M)	<Cpo_____> = TransAC <TransBC <TransAB p>>$_M$	
2	A/B * B/C ⇒ A/C	<Spec_____> = IntAC <IntBC IntAB, (p,x)>$_C$	
3	A/B * B→C ⇒ A/C (M)	<TransBC IntAB>$_M$ = IntAC	ordinary computation
4	A→B * B/M ⇒ A/M (M)	<Cpo_____> = IntAM <IntBM <TransAB p> x>$_M$	translator to interpreter
5	B/C ⇒ B→C (C)	<Spec'_____> = TransBC <Spec_____p___>$_C$ <IntBC •,x>$_C$	interpreter to translator

References

[1] Bourbaki N., Éléments de Mathématique. Théorie des Ensembles. Vol. Premier Partie, Livre I, Hermann 1960.

[2] Fegaras L., Sheard T., Zhou T., Improving programs which recurse over multiple inductive structures. In: ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation. 21-32, 1994.

[3] Futamura Y., Partial evaluation of computation process - an approach to a compiler-compiler. In: Systems, Computers, Controls, 2(5): 45-50, 1971.

[4] Glück R., Towards multiple self-application. In: Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation. (New Haven, Connecticut). 309-320, ACM Press 1991.

[5] Glück R., Klimov A. V., Occam's razor in metacomputation: the notion of a perfect process tree. In: Cousot P., Falaschi M., Filè G., Rauzy A. (eds.), Static Analysis. Proceedings. (Padova, Italy). Lecture Notes in Computer Science, Vol. 724, 112-123, Springer-Verlag 1993.

[6] Glück R., Klimov A. V., Metacomputation as a tool for formal linguistic modeling. In: Trappl R. (ed.), Cybernetics and Systems '94. Vol. 2, 1563-1570, World Scientific: Singapore 1994.

[7] Jones N. D., Gomard C. K., Sestoft P., Partial Evaluation and Automatic Program Generation. Prentice Hall International Series in Computer Science. Prentice Hall: New York, London, Toronto 1993.

[8] Turchin V. F., The Phenomenon of Science. Columbia University Press, 1977.

[9] Turchin V. F., The concept of a supercompiler. ACM TOPLAS, 8(3): 292-325, 1986.

[10] Turchin V. F., A constructive interpretation of the full set theory. In: The Journal of Symbolic Logic, 52(1): 172-201, 1987.

[11] Turchin V. F., On cybernetic epistemology. In: Systems Research, 10(1): 3-28, 1993.

[12] Wadler P., Deforestation: transforming programs to eliminate trees. In: Theoretical Computer Science, 73: 231-248, 1990.