KEIM: A Toolkit for Automated Deduction

Xiaorong Huang Manfred Kerber Michael Kohlhase Erica Melis
Dan Nesmith Jörn Richts Jörg Siekmann

Fachbereich Informatik, Universität des Saarlandes
66041 Saarbrücken, Germany
`keim@cs.uni-sb.de`
Telephone: (49) 681-302-4627

Abstract. KEIM is a collection of software modules, written in Common Lisp with CLOS, designed to be used in the implementation of automated reasoning systems. KEIM is intended to be used by those who want to build or use deduction systems (such as resolution theorem provers) without having to write the entire framework. KEIM is also suitable for embedding a reasoning component into another Common Lisp program. It offers a range of datatypes implementing a logical language of type theory (higher order logic), in which first order logic can be easily embedded. KEIM's datatypes and algorithms include: types; terms (symbols, applications, abstractions); unification and substitutions; proofs, including resolution and natural deduction styles.

1 Motivation

Though automated reasoning systems are among the earliest AI programs, the methods developed and implemented by the theorem-proving community are little-used outside of it. This phenomenon may be explained to some extent by the computational complexity of the programs involved (often NP-complete or undecidable), but an even larger share of the blame may be assigned to the *cognitive* complexity involved in the implementation of the programs themselves. It is easy to describe a proof process such as resolution, but actually writing a fairly-efficient resolution prover is far from trivial. In addition, a prover requires subcomponents, such as formula parsing and pretty-printing, which add to the magnitude of the job. The work and experience required to build a good theorem prover from scratch can be daunting for an outsider. Especially when the theorem prover is not the main object of study, but rather intended to be used as a component in some larger system, the foreseen difficulties will discourage many from beginning.

One may of course decide to use a prover that is already available. This has the advantage that its reliability is relatively assured and, being off-the-shelf, requires no implementation. Unfortunately, it is rare that the needs of a new application exactly fit the strengths of an existing theorem prover. Even if that were the case, one would probably have to build some kind of bridge between the two programs in order to exchange data, because the basic data structures used (terms, formulas, etc.), not to mention input syntax, are probably incompatible.

One may try to modify the source code directly, but this is a very difficult task for the nonexpert.

In addition, most theorem provers are *sui generis*; they are designed to investigate a particular paradigm or approach and are not intended to be useful for all types of reasoning problems. This limits their applicability among a wide audience. And trying to get two provers to cooperate without greatly changing at least one of them is not a task for the faint of heart.

Because of these difficulties, those who wish to apply techniques developed by the theorem-proving community face the choice of either learning this 'black art' themselves by developing their own prover from scratch, or jury-rigging available provers to get some kind of result. Hardly an encouraging prospect. Even automated reasoning experts may wish to make a theoretical study of just a minor aspect (say, a comparison of term indexing schemes), and not want to go to the trouble of implementing the whole environment normally required.

We feel that what is needed to make theorem-proving technology widely available in a useful way is a framework that provides the essential tools (data structures and algorithms) to allow a theorem prover to be assembled by a non-expert. Such a framework must be modular, to allow data structure or algorithm variants to be swapped in or out, and extensible, to permit customization, as well as the addition of new modules, with relatively modest effort.

2 An Open Architecture

Despite the diversity of theorem provers currently in use and in development, there are many aspects that they share. They must support basic data structures such as terms, formulas and, often, more complex objects such as clauses and substitutions. There must be a way to parse user input into a usable form, and to pretty-print the internal format in a human-readable way. Unification and/or matching are also common components.

There are well-known algorithms and techniques for each of these areas. Their current implementations, however, are not suited for generic use, often relying on varying idiosyncratic data structures which cannot be reconciled. It is certainly necessary to continue research in the optimization of techniques such as unification, but for many applications, choosing one of the currently-known variants is good enough. Most users of theorem-proving technology do not want to reinvent the wheel, and even the best-known algorithms may be difficult to implement correctly and efficiently.

KEIM provides a framework, through documented interface functions, which allows such techniques to be implemented in a generic way, so that later improvements or customizations can be carried out without requiring changes to other modules. Just as a toolbox holds several similar tools which do roughly the same thing, KEIM will contain differing implementations of data structures and algorithms. These will be provided in a modular form that will allow unnecessary components to be left out and improved components to be swapped in.

KEIM is like a cafeteria of theorem-proving tools, providing wholesome options that can be appetizingly combined.

This modularity of KEIM is essential for three reasons. First, the state of knowledge in theorem proving is always expanding; there will always be new ideas and techniques worthy of sharing. Second, the resources of any one group are limited. There is no way a single group can be expert in all the aspects of theorem proving that KEIM should offer. The cooperation and contributions of others must be possible if KEIM is to be truly useful. We wish KEIM to be a toolbox, not a toybox with only sketchy or incomplete implementations of certain techniques. Last but not least, it will be the rare case that a user will want to use exactly what is provided without any customizations. This is especially true when a theorem prover is to be embedded in an existing system, with its own data structures. Tools that solve the wrong problem would be of little use.

3 The KEIM Toolbox

KEIM version 1.2 [3] is implemented in Common Lisp, using the Common Lisp Object System (CLOS) [5]. CLOS allows great flexibility in the integration of new classes of objects. The generic function paradigm allows one to specialize the behavior of a function on a new type of object without changing its behavior on existing objects and without having to rewrite or copy existing and unrelated code, thus making it well-qualified for the implementation of modular, extensible toolboxes.

KEIM offers a range of datatypes implementing a logical language of type theory (higher order logic) called $\mathcal{POST}$ [3], in which first order logic can be easily embedded. KEIM's datatypes and algorithms include: types; terms (symbols, applications, abstractions); unification and substitutions; proofs, including resolution and natural deduction styles.

KEIM also provides functionality for the pretty-printing, error handling, formula parsing and user interface facilities which form a large part of the code of any theorem prover. These facilities are all easily customizable.

KEIM serves as the basis for the Ω-MKRP [2] proof development environment (successor to the MKRP project), and is developed as part of the German Deduction Effort, which is sponsored by the Deutsche Forschungsgemeinschaft as "Schwerpunkt Deduktion".

Cooperation with other research groups is underway, and a KEIM implementation of the ACID [1] term indexing software has been made available.

3.1 A Scenario

Suppose a user wishes to build a small resolution prover. She must first decide what logical language the prover will allow by selecting the KEIM modules that contain the corresponding CLOS class definitions. She may want to make some minor adjustments—to the pretty-printing functions, for example. Another change may be to specialize a class, *e.g.*, she may want to add a slot to the clause class which counts the number of times it was used. This will then require adding a method to the generic function that actually does the resolution, so that the method updates the slot.

Modules will be chosen for the desired types of resolution, factoring, etc. If the prover is to be interactive, commands can be defined in Lisp using the KEIM primitives. Some Lisp ability will be required to sew things together. The modules are loaded in the proper order into a Lisp environment, and the prover is ready for action. An example of a simple tableau prover and resolution prover written in KEIM are described in [4].

4 Summary and Future Directions

KEIM is a software project that intends to offer, through its software library, a way for the general AI community, as well as the theorem-proving community, to take advantage of the many developments that have been made in automated reasoning. KEIM will provide standard implementations of many techniques and algorithms, making it useful not only for building reasoning systems, but also for pedagogical purposes. KEIM is available for anonymous FTP from various locations; send e-mail to `keim@cs.uni-sb.de` for instructions.

We intend to extend KEIM in both breadth and depth, that is, both to improve the current implementations, as well as to add new variants (*e.g.*, various unification algorithms, equality-handling mechanisms). Another goal is to make KEIM even easier for nonexperts to use by providing a better user interface. Because of the amount of KEIM code, it can be intimidating for those who are just starting. We want to make getting started with KEIM a painless process.

We intend to explore cooperation with other groups who have expertise in particular areas, and welcome collaboration and suggestions. Currently we are setting up a KEIM user group consisting of those who do some implementation on the basis of KEIM.

KEIM's extensibility and customizability is intended to make it an open architecture for (Lisp-based) reasoning systems. We hope that KEIM, by providing the building blocks of a reasoning system, will allow others to concentrate on the research areas which are of principal interest to them.

References

1. P. Graf: Path Indexing for Term Retrieval. Technical Report MPI-I-92-237, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1992.
2. X. Huang et al.: Ω-MKRP, A Proof Development Environment. In these proceedings.
3. D. Nesmith, editor: KEIM-Manual version 1.2. Universität des Saarlandes, Im Stadtwald, Saarbrücken, Germany, 1994.
4. J. Richts and D. Nesmith: Implementing Simple Theorem Provers in KEIM: Case Studies. To appear as a SEKI Report, Universität des Saarlandes, Im Stadtwald, Saarbrücken, Germany.
5. G. Steele: Common Lisp, second edition. Digital Press, Boston, 1990.