Canonicalization in the PrIKL Reasoner

Don Libes, Antoine Gerardin, Severin Tixier, and Fabian Neuhaus
National Institute of Standards and Technology, Gaithersburg, MD, USA
Email: {libes,gerardin,tixier,fneuhaus}@nist.gov

Abstract–**Many objects in the PrIKL Reasoner have a canonical representation. This representation and its implementation required a number of choices with ramifications. This paper describes the choices, their consequences, and some of the more interesting implementation details. This information will be helpful in understanding the implementation of PrIKL, particular for implementors of PrIKL plugins. It may also be helpful in designing systems with similar representational problems.**

Index Terms–**canonical representation, reasoner, prover, PrIKL, IKL**

I. Background

PrIKL[1] is software to reason with logical statements using a hybrid of a natural deduction system and a production system [1][2]. PrIKL is based on the IKL knowledge representation language and Common Logic family of logic languages [3][4]. PrIKL includes additions to first-order classical logic of sequence variables and a way to reason about propositions. PrIKL is implemented in Java[2] [8].

PrIKL is being developed as part of a semantic validation suite for information flows in manufacturing production networks. The development and production of complex products demands the cooperation of engineering and operations activities across a network of companies, and the successful exchange of information among them. Standards assist in the mechanics of information transfer but do not ensure *fitness for purpose*: consistency, completeness, and timeliness of the information. The semantic validation suite will enable testing for fitness of purpose by translating the content of messages, metadata about the messages, as well as background information into an IKL knowledge base.

II. Introduction

We intuitively think of logical statements as objects that have structure. For example, a quantified sentence has a quantifier, binding variable, and a sentence. Each of these in turn have structure. For example, the sentence may itself be a quantified sentence.

Internally, PrIKL generates new sentences and performs tasks such as comparisons to existing sentences and unification with other sentences. Conceivably, these tasks could all be accomplished using this structured representation. For example, a function to do comparison would look like:

```
Boolean b = equal(Sentence s1, Sentence s2);
```

At a minimum,this type of test for equality would require a parallel tree traversal as the objects in both sentences are compared.

However, as the ratio of comparison tests to object modifications grows, repeated tree traversal of unchanging trees becomes expensive. An efficient representation is needed. We address this with a string representation that effectively appears to be an IKL string representation.

An additional requirement is the ability to maintain collections of sentences. The collections must provide a way to look up sentences or iterate over the sentences. We address this with dictionaries keyed by the string representations.

A benefit of the IKL-based string representation is that users can readily see objects in a form that makes sense. While this could be viewed as a requirement, it is not a requirement in the sense that reasoning can take place without human interaction. Only in the final step (or during debugging) might it be helpful to provide a string representation.

III. String Representation

The string representation of a logical sentence is an obvious alternative representation of a sentence object. Indeed, input to the system starts with string representations that are parsed into tree structures. However, after parsing, the string representations are discarded since they are not in a consistent form. For example, consider:

```
(if(P)(Q))
(if (P) (Q))
```

Both sentences are equivalent but their string representations are different. An intelligent string comparator

1. PrIKL is an acronym for Prover for IKL. IKL is an acronym for IKRIS Knowledge Language. IKRIS is an acronym for Interoperable Knowledge Representation for Intelligence Support.
2. Any mention of commercial products is for information only; it does not imply recommendation or endorsement by NIST.

doi:10.4304/jsw.8.5.1058-1062

could recognize these are the same; however, it is simpler to just generate the string anew because that guarantees the string representation is in a standard form. This standard representation is known as Prikl's canonical form. For short, we call this a *canonical*.

PrIKL users do not need to know the peculiarities of the canonical or how it is managed internally. However, anyone debugging or expanding PrIKL (e.g., plugin authors) must understand the canonical and how it is managed. The canonical is ultimately shown to the user during the explanation phase of the reasoner. Of course, the canonical form is compatible with PrIKL's own parser.

IV. Dictionaries

PrIKL maintains objects in a variety of dictionaries. For example, during reasoning, storage of sentences is provided in a dictionary known as the *magic box*. A key aspect of dictionaries is that objects can be retrieved by their key. For most dictionaries in PrIKL, the key is the object's canonical.

For example, as each new assumption is generated, it is important to know whether it matches any of a variety of target sentences that would end the proof – e.g., by raising a contradiction. This is accomplished in PrIKL by using the canonical of each new assumption as a key to find whether any sentence already appears in the magic box with that key.

In the particular case of a request to add a contradictory assumption, this is detected by finding the negated assumption in the magic box. That is, when a new assumption is encountered, both it and its negation are added to the magic box. (Of course, the negation is flagged to indicate it is not an assumption.) Intuitively, an additional flag within each sentence could itself declare which version of the sentence is being asserted. Indeed PrIKL sentences do have such a flag. But for the lookup purposes, there are a number of reasons why the flag does not offer a big payoff. The simplest reason is that every test for existence would then have to have a subsequent test for negation. A more complex issue is that the existence of a sentence or its negation could trigger (future) or track (history) two entirely different chains of reasoning. Rather than stuff both into a single dictionary entry, we believe it simpler to make separate entries. This decision may be worth revisiting in the future.

Although lexical negation – the nesting of a sentence inside of a "(not ...)" – is simple, many sentences in PrIKL do not syntactically resemble their negations because PrIKL's focus on natural deduction pushes negations into sentences rather than leaving them at the outside, if possible.

This use of dictionaries was an implementation choice that bears closer scrutiny as its advantages and disadvantages are not as trivial as they would first appear. And there are alternatives. For example, rather than using an explicit canonical, the collection of all sentences in the system could be maintained as a tree. The first level of the tree could be the negation type. The next level could be the sentence type (quantified, boolean, atomic), and so on.

However, at some point in a tree, the structure bottoms out, leaving a problem similar to the original one (i.e., that the magic box addresses). For example, consider atomic sentences such as (P a), (P b), and (Q b). These can be placed in the tree under a positive/negative node followed by an atomic node but what then? Do all atomics get stored in, say, a positive-atomic list or hash table? If a hash table, is it split into separate hash tables for predicate? Or split again by terms and the number of terms? Since there are tradeoffs at this level, is an adaptive approach appropriate?

It is also worth mentioning that the dictionary provides performance of O(1) compared to O(n) for trees where n is the complexity of the sentences rather than the size of the problem. The size of the problem should be a factor but we omit it here because it adds similar complexity to both approaches.

Although we characterized hashing as O(1), of course that is idealistic. This is not the paper to expand on that. However, it is worth noting that the default Java string hashing executes m multiplications, one for each character in a string. If many sentences are lengthy then, the hashing cost can be significant. In addition, our canonical form bears significant redundancy. For example, quantified statements all begin with "(forall (". At the very least, it would make sense to use a hashing algorithm that is smart enough to skip over areas of likely redundancy. We plan to address this in the future. At the same time, the natural deduction aspect of PrIKL drives new sentences to be shorter over time. While there are exceptions, we expect performance to be acceptable because sentences generally shrink rather than grow.

V. Memory

PrIKL's canonicalization is extravagant when it comes to memory use, an outcome of a *memory is cheap* philosophy. We believe that our use of this philosophy is appropriate but we will revisit it in the future. In the meantime, we provide some details about memory use in this section.

A canonical is generated for each object. This is true not only at the sentence level but for each component of the sentence, recursively. There is in effect a pyramid of canonicals supporting each sentence.

The intensive memory use is somewhat ameliorated by several factors:

- There is an extensive amount of sharing. For instance, the canonical of a binding variable is shared by all references to the binding variable. Sentences (and subsentences) themselves may also be shared by other sentences.
- Memory is cheap. This philosophy is used throughout PrIKL so it makes sense to be consistent during canonicalization as well. At the same time, PrIKL's memory use is tempered by problem splitting – the

splitting of problems which are then offloaded to other machines.

- The pyramid of canonicals can be lessened by skipping levels and recomputing canonicals as necessary. In effect, this trades back memory for CPU time. For historical reasons, level skipping currently occurs in the system for sentences lists. Such sentence lists occur only in boolean sentences. (Sentence lists do not have a specific object type but are merely implemented as raw Java lists. In retrospect, it was a mistake to avoid the explicit object; however, for canonicalization purposes, this would be one of the levels where canonicalization would be skipped anyway.)
- Whitespace is skipped whenever unnecessary. This saves space although it makes canonicals look less like they were written by a human. It might be appropriate to rewrite canonicals before they are ultimately shown to the user during the explanation phase of the proof. At the same time, it is worth noting that IKL is a verbose language. For example forall takes six characters when it could be replaced with a single Unicode character [7]. Conceivably, we could do the opposite action – rewriting to use traditional characters because users would prefer seeing the simpler and shorter sentences.

VI. Canonicals On Demand

Canonicals do not necessarily exist for all objects; only those for which the canonical representation is required. In addition, objects that are modified do not necessarily have their canonicals updated. Again, recanonicalization only occurs when the canonical is needed.

When an object's getCanonical method is called, the system calls isValidCanonical. If the canonical is invalid, it is regenerated using the canonicals of the components, recursively. This recursion stops when isValidCanonical is true.

If object modifications are made that affect the canonical, objects must call invalidateCanonical. Certain types of modifications require the entire pyramid of canonicals be recomputed in which case invalidateAllCanonicals must be called.

An earlier version of PrIKL had no such canonical maintenance. Programmers implementing or extending PrIKL had to decide when to regenerate canonicals. Since the system made no promises, implementors found themselves regenerating canonicals after each object modification or taking a significant risk that canonical regeneration could be deferred safely. Since the implementors had no idea whether there would be a need for the canonical, regenerations were frequently thrown away before ever being used and were replaced with a new canonical. The current on-demand system is a significant performance improvement on the original system. The new system relieves the implementors of having to worry about whether canonical regeneration could be deferred.

VII. Names

The canonicals of primitive objects such as variables and constants are their names. For example, the canonical of the IKL name "foo bar" is ""foo bar" - with the quotes.

It is counterintuitive to include the quotes; however IKL distinguishes between certain objects with punctuation. Specifically, unquoted strings and double-quoted strings are IKL *names* while single-quoted strings are IKL *char strings*. PrIKL maintains a dictionary of such objects which are closed terms. Because the canonicals include the quotes, the canonicals can be used as dictionary keys directly thereby avoiding the seeming ambiguity between 'foo bar' and "foo bar".

IKL names require double quotes only if the names include special characters such as whitespace. This raises the ambiguity that "foo" is equivalent to foo. To avoid this ambiguity, PrIKL removes the quotes when possible.

PrIKL names new objects or renames old objects during reasoning. Since IKL does not permit quoted names to begin with a backslash, all names PrIKL introduces are prefixed with a backslash to prevent confusion or pollution of the user namespace. In the future, we will consider using namespaces to avoid the extension to the IKL specification.

The following table defines the prefixes created during renaming:

\q	quantifier names
\x	term names
\sk	skolem function names
\seq	sequence variable names

The original name is appended by a unique integer (e.g., \q0) or with the user's original name optional inserted as well (e.g., \x_person_17). Note that PrIKL reserves additional prefixes for other purposes. For example, \s is used for sequence variable name prefixes created during unification – but this has nothing to do with canonicalization.

VIII. Quantifiers

Quantifiers receive special handling during canonicalization.

- Users provide common quantifier names such as "x" in (forall (x) ... It is not unsurprising (and is legal) to have nested quantifiers with the same name. To avoid redundancy of quantified sentences, all quantifiers are renamed systematically. For efficiency, the most deeply nested quantifiers are renamed first. This 'inside-out' method avoids having to recanonicalized subsentences as they appear in new sentences. For example, both of the following sentences are logicially equivalent:

```
(forall (x y) (and (P x) (exists (x) (Q x y))))

(forall (man woman) (and (senseOfHumor man) (exists
(child) (knows child woman))))
```

Both of those sentences have the same canonical:

```
(forall (\q2 \q1) (and (P \q2) (exists (\q0) (Q \q0 \q1))))
```

During transformation to conjunctive normal form, two actions are taken which again affect canonicals:

- Universal quantifiers are dropped and variables are freed.
- Existential quantifiers are dropped and variables become skolem functions.

In both of these cases, unique variables names are generated to avoid collisions with other names. In the case of skolem functions, references to the same variables within the scope of the function are given the same names across the entire sentence. For example:

Before skolemization:

```
(and
    (forall (\q1)
        (or
            (not (P \q1))
            (exists (\q0) (P \q1 \q0))))
        (forall (\q1)
            (or
                (forall (\q0) (not(P \q1 \q0)))
                (P \q1))))
```

After skolemization:

```
(and
    (or
        (not (P x_5))
        (P x_5 (skolem x_5 "\sk_y_1")))
    (or
        (not (P x_7 y_6))
        (P x_7)))
```

IX. Public API

The canonicalization API is defined by the SimpleNode abstract class. (For historical reasons, part of the API appears in the TreeElement interface. This should eventually be reorganized.)

The primary public methods to retrieve a canonical is getCanonical. It takes no arguments and returns the canonical, generating it as necessary. A deprecated method of the same name takes a CanonType argument. It is mentioned here because many uses of it remain in the code. CanonType selects between:

CT_ALL	Regenerate all levels.
CT_MIN	Generate the minimum number of canonicalizations.
CT_CURRENT	Force canonical regeneration at the current level.

Two methods are supplied to tell the system that the object has changed and that the canonical is invalid:

```
invalidateCanonical()
invalidateAllCanonicals()
```

Once a modification has been made to an object that causes its canonical to become invalid, invalidateCanonical indicates that the cached canonical at the current level is no longer valid. invalidateAllCanonicals causes the entire pyramid of canonicals to be invalidated so that any request for a canonical starts the process from the leaves. This is necessary for certain changes such as wrapping a sentence in a new quantifier. Typically, quantifier manipulation is more complex and it is simpler to recreate the quantifier than track and adjust all the references to its canonical.

The following method is useful in unusual cases:

```
getInvalidCanonical()
```

It is sometimes necessary to get the canonical of an object even though it is known to be invalid. Obviously, this method must be used with care as there are no guarantees regarding how the object has diverged from its canonical. For example, getInvalidCanonical is used when terms are declared identical (e.g., (= a b)) and such identities are being propagated through sentences which use the terms in the identity. In this situation, old sentences must be marked appropriately in the magic box. However, the only way to get to the sentence entries in the magic box is by their canonicals (i.e., keys). Calling getCanonical runs the risk of regenerating the canonical but that would result in a new canonical when the original is needed. Hence, getInvalidCanonical is used in this situation.

The following methods are used for logging and debugging:

```
toString()
toDetailString()
getOriginalString()
```

toString is a part of the Java contract for all objects although no guarantee is made of its format. However, toString is used by IDEs such as Eclipse and Netbeans, so we provide an implementation with a useful representation [5][6]. Depending on debugging desires, toString can be defined to return either getCanonical or getDebugCanonical.

toDetailString is a method that returns a string with additional details such as parser token identification.

getDebugCanonical does not return a canonical but rather a version based on the user's original input. The version is not likely to be exact because no attempt is made to track whitespace and other miscellany. However, it returns original object names before they have been changed by canonicalization as described elsewhere. Objects that have been created out of thin air by the system (e.g., skolem functions) use their canonical when queried for their original string.

For efficiency and simplicity, *debug canonicals* are created at the same time as the regular canonicalization. This doubles the memory usage of the canonicalization. However, these debug canonicals may be turned off (in util.Debug.debugging), in which case debug canonicals are not

generated. References to them will generate a simple diagnostic message but will not raise an error.

If the user has asked for binding information, this is generated in the final step of a report. Another function is provided for this purpose: getOriginalString is similar to getDebugCanonical except that getOriginalString 1) rebuilds canonicals from scratch each time and 2) provides more original information than getDebugCanonical. getOriginalString almost always produces ambiguity – for example, two variables may have originated as a a single variable – so it should generally be avoided. getOriginalString is also used as a convenience for writing unit tests of the system itself as deducing the correct numbering for canonicals is often not the point of many unit tests. Thus, implementors do not have to deal with the extra layer of indirection caused by renaming during canonicalization.

X. Implementation API

This section describes the API used in maintaining canonicals. It will necessarily make references to Java techniques since that is how the system is implemented.

Canonicalization requires two passes through an object tree. The first pass computes the depth of objects where each additional binder increases the depth. The innermost binder is depth 0. Sentences may have multiple depth-0 binders in different branches of a boolean.

The two passes are necessary because 1) the outer sentences cannot have a canonical representation until the inner sentences have had their depth computed, and 2) inner sentences cannot have their canonical created until the binders in outer sentences have had their canonicals created. Two separate passes avoid this dilemma.

In the first pass, canonicalization starts with a test of isValidCanonical which is a marker that exists within every object recursively. If the canonical is valid, computation stops and the previously cached canonical is returned. As the object is traversed using calculateCanonicalDepth, setCanonicalDepth is called to record the computed depths.

In the second pass, cannonizeWithoutDepth descends through the object tree (guided by isValidCanonical as in the first pass) building the canonical using the depths computed in the earlier pass available through getCanonicalDepth. As canonicals are finished, each object is marked with validateCanonical, which plays the role of setter for isValidCanonical.

One last significant method is getSkolemName which returns a name appropriate for use within skolem functions. If the term has never been used for this purpose, a new skolem name is generated using genSkolemNameString, otherwise its existing skolemNameString is returned for this purpose.

Some additional methods are used for miscellaneous purposes. For instance, wrapSentenceCanonical takes a sentence and generates the canonical aspects common to all sentences such as outer parentheses and negation. There are many name generator-related functions that shall not be mentioned here but can be found in the source [2].

XI. Comments

User-provided comments are accepted by the PrIKL parser. However, they play no role in reasoning. Conceivably, they are of use to users at the conclusion of the proof and can be provided in the canonical. Thus, user comments are only provided in the debug canonical.

XII. Summary

Canonicalization plays a crucial role in the PrIKL reasoner. Canonicalization is needed for both object access as well as explanations to the user. Canonicalization is complex and has subtleties including tradeoffs given expectations governing CPU, memory, and problem size. We have made decisions about these tradeoffs and experimented with implementations arriving at our current system. We expect that canonicalization will bear continued scrutiny and experimentation in the future.

References

[1] Neuhaus, F. et al., PrIKL, A Natural Deduction Reasoner, in preparation. Contact fneuhaus@nist.gov for availability.

[2] PrIKL source, http://prikl.sourceforge.net.

[3] IKL Specification Document, http://www.ihmc.us/users/phayes/IKL/SPEC/SPEC.html, retrieved Jan 10, 2012.

[4] Common Logic Standard, http://iso-commonlogic.org, retrieved Jan 10, 2012.

[5] Eclipse, http://www.eclipse.org, retrieved Oct 1, 2011.

[6] Netbeans, http://netbeans.org, retrieved Oct 1, 2011.

[7] The Unicode Standard: A Technical Introduction, http://www.unicode.org/standard/principles.html, retrieved Oct 1, 2011.

[8] Java, http://www.java.com, retrieved Oct 1, 2011.