Pascal-F Verifier Internal Design Document

CPCI #4 -- Simplifier

John Nagle

FACC / Palo Alto

1. Introduction

The simplifier is a modified form of the simplifier developed by Dr. Derek Oppen while at Stanford University. A rule handler and type handling have been added, and these are documented here.

2. Rule handling

2.1 Interface with the Boyer-Moore prover

When the simplifier is invoked, it reads the file "ruledatabase", in which it expects to find Boyer-Moore theorem prover events in the form extracted by the "getevents" program. Rules are PROVE.LEMMA events with a usage of (REWRITE) and a name ending in ".rule" or ".RULE".

2.1.1 Validation of rules

There are certain requirements imposed on rules to insure that an inconsistency is not introduced in the simplifier. First, the top operator of any rule must be Boolean-valued. Second, all DEFN and DCL functions in the rule must be of known type.

2.1.2 Validation of DEFN definitions

Only definitions whose results are Boolean or integer may appear in rules. The type of the DEFN is obtained from the Pascal-F definition of the rule function as expressed in the Jcode. Functions, of course, are restricted to simple return types in Pascal. This is very valuable to us because the problem of determining the type of an array-valued DEFN is difficult. DEFNs are scanned during the reading of the "ruledatabase" file and an attempt is made to determine the type of the DEFN. If the top operator of the DEFN is an operator which is always integer or always boolean, we are in good shape and thus know the type of the DEFN. If the type operator is IF, we examine the arguments to the IF, and insist that both possibilities be the same type. If this process is unable to determine the type of the DEFN, we look for a PROVE.LEMMA event of the form

```
(integerp! (f X))
```

or

```
(booleanp! (f X))
```

and if one is found, we then know that the function f is of the given type. Failure to find a type by either means for any function mentioned in a rule is a fatal error. The type determined from this analysis is checked for compatibility with the rule function declaration and a difference is a fatal error.

3. Type handling

The original Oppen prover had no knowledge of types. We have found it necessary to add extensive knowledge about type information to make the theory of the prover consistent with that of the Boyer-Moore prover.

3.1 Type descriptions

Types are S-expressions as described in the Jcode document. We also allow

(integer)	The most general form of numeric type.
(universal)	Any type will match this.
nil	No type will match this; generated when types clash.

3.2 Types of operators

Almost all operators return constant types. All the arithmetics return *integer;* all the booleans and relationals return *boolean.* The array and record operators return results appropriate to the object being operated on.

3.3 Types of variables

Variables and non-builtin functions are typed given the type information provided with the **vardecl** call.

3.4 Type predicates

The following type predicates are understood in formulae:

(arrayp! X)	True if X is an array of any type, false otherwise.
(integerp! X)	True if X is an integer, false otherwise.
(numberp! X)	True if X is a nonnegative integer, false otherwise.
(booleanp! X)	True if X is a boolean, false otherwise.

3.5 Implementation of types

3.5.1 Additions to the Equality theory

Every enode in the E-graph has a type value, called *edatatype.* The type of a node is considered to be the type of the root node of its equivalence class. The getdatatype function in the prover will return the type of a node by retrieving the edatatype of the base node.

Whenever the type of a node is changed, the type and node are pushed on the context stack along with the function *popdatatype* which will restore the old type after a popcontext.

When two nodes are merged, the resulting type is the minimal common subtype of the types of the nodes being merged, as computed by *commontype.* If this is nil, *propagatefalse* is called to cause the present propositional case to be aborted as unsatisfiable.

Enode requires some additional processing. The operators *selecta!, storea!, selectr!, storer!,* and *alltrue!* have result types depending upon the types of the operands. For all other operators, the type is obtained from the ’type property of the function atom. Return of a null type is an error.

3.5.2 Additions to the normalizer

The *typed!* dummy function is used to transmit the types of variables to the simplifier proper. The*typed* function should appear around every reference to every variable in the input formula. *typed!* is no longer applied to expressions.

3.5.3 Type predicates

Type predicates are implemented as demons which examine the type and merge the given node with true or false as required, except for "numberp!", which generates a form to test if the value is nonnegative. The form generated is "(equal! node (gei! X 0))", and this is applied with emerge, not propagate, so that it is evaluated immediately. This approach to type predicates prevents excessive casing.

3.5.4 Type restrictions on the subtheories

There are basic inconsistencies between Boyer-Moore theory and Oppen theory. In the original Oppen prover,

    (equal! (addi! X 0) X)

is an axiom. In the Boyer-Moore prover, using Boyer-Moore theory of Z-numbers,

```
(IMPLIES
   (NOT (integerp! Y))
   (EQUAL (addi! X Y) 0))
```

is a theorem. This leads to the contradiction that

```
TRUE + 0 = TRUE          in Oppen
TRUE + 0 = 0             in Boyer-Moore
```

which can be misused to prove TRUE = FALSE, leading to unsound verifications.

All the type machinery is installed primarily to prevent this conflict. The approach used is to prevent the operation of the decision procedures of the Oppen prover on objects of the wrong type. The two trouble spots are the Oppen theory of numbers (Z) and the Oppen theory of arrays (A). Our type theory allows us to know the type of any node in the formula before beginning work on that node. We must not submit an arithmetic operator to the Z-prover unless all operands are of type integer, and we must not apply the rules for arrays unless the operands are of the proper types.

The interface between the Z-prover and the E-graph where information goes to the Z-prover is in *enodenonatomic.* A check is needed there to insure that all operands are numeric. Where operands are not numeric, this is not an error, but the operator must be treated as an uninterpreted function.

The array rules are easier, because each has its own demon. We simply add type checking to the demon.