Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 196 (2008) 129–135
www.elsevier.com/locate/entcs

Signature Compilation for the Edinburgh Logical Framework[1]

Michael Zeller, Aaron Stump, and Morgan Deters

Computational Logic Group
Computer Science and Engineering Dept.
Washington University in St. Louis
St. Louis, Missouri, USA

Abstract

This paper describes the Signature Compiler, which can compile an LF signature to a custom proof checker in either C++ or Java, specialized for that signature. Empirical results are reported showing substantial improvements in proof-checking time over existing LF checkers on benchmarks.

Keywords: Edinburgh LF, signature compilation

1 Introduction

The Edinburgh Logical Framework (LF) provides a flexible meta-language for deductive systems in several application domains [1]. A well-known example is for proof-carrying code [2]. Another example is for proofs produced from decision procedures [5]. A single LF type checker can be used to check proofs in any deductive system defined by an LF signature (a list of typing declarations and definitions). LF implementations like Twelf work in an interpreting manner: first the LF signature is read, and then proofs can be checked with respect to it [4]. This system description (LFMTP 2007 Category C) describes the Signature Compiler ("sc") tool, which supports a compiling approach to LF type checking: an LF signature is translated to a custom proof checker specialized for that signature. Signature compilation emits checkers that run much faster than existing interpreting checkers on benchmark proofs, as shown in Section 3, including proofs produced by a QBF solver for QBF benchmark formulas. The Signature Compiler is publicly available

[1] This work supported by the U.S. National Science Foundation under grant numbers CCF-0448275 and CNS-0551697.

doi:10.1016/j.entcs.2007.09.022

```
o : type.     trm : type.

== : trm -> trm -> o.     imp : o -> o -> o.

%infix left 3 ==.    %infix left 5 imp.

pf : o -> type.

impi : {p:o} {q:o} (pf p -> pf q) -> pf (p imp q).
mp : pf (P imp Q) -> pf P -> pf Q.
```

Fig. 1. Fragment of example LF signature

from the "Software" section of http://cl.cse.wustl.edu. For space reasons, this paper must assume familiarity with LF and its Twelf syntax.

2 The Signature Compiler

The intended use of sc is for generating backend checkers, which are optimized for the case when the proof successfully checks. Thus, sc does not report useful error information for failed proofs. Also, backend checkers allow (untrusted) proofs to contain additional definitions, but not additional declarations, which might subvert the deductive system defined by the (trusted) signature. The ideal case for use of sc is when many proofs expressed with respect to the same signature need to be checked efficiently. In such a case, reuse of the custom checker generated by sc makes up for the time needed for signature compilation.

The Signature Compiler parses an LF signature in Twelf syntax, and generates all the source files required for a proof checker that checks proofs expressed with respect to that signature. The Signature Compiler supports fully explicit LF in Twelf syntax, without type-level λ-abstractions (a common restriction, not essential for sc), and where constants declared in the signature must be fully applied when used. The checkers emitted by the Signature Compiler, but not sc itself, also support a form of implicit LF, in which holes ("_") can be written in place of arguments to constants c from the signature, as long as the values of those holes can be determined by unification in the higher-order pattern fragment from the types of other arguments to c. Support for more aggressive compression schemes must remain to future work (cf. [3]). The Signature Compiler is written in around 3000 lines of C++ and can generate custom checkers in both C++ and Java.

Figure 1 gives part of a standard LF signature for an example logic with equality, implication, and universal quantification. This logic is used for the benchmarks below. For space reasons, the figure focuses just on implication; see the Appendix for the complete signature. Infix directives in Twelf syntax, used after the declaration of "`imp`" in the figure, are supported by Signature Compiler, and by the emitted checkers.

The Signature Compiler emits code for custom parsers for each signature it compiles. Neither the emitted parsers nor sc itself relies on parser or lexer generators, since such reliance would increase the size of the trusted computing base, and make it more difficult to support infix directives in proofs. Simple lexer generation – in particular, creating an inlined trie – is performed by sc for lexing efficiency in the

```
case /*===*/ X61o61o_EXPR: {
  /*===*/ X61o61oExpr *e = (/*===*/ X61o61oExpr *)_e;
  if( (areEqualNuke(computeType(e->e1),
                    new /*trm=*/ XtrmExpr()) &&
       areEqualNuke(computeType(e->e2 ),
                    new /*trm=*/ XtrmExpr()) ))
    return new /*o=*/ XoExpr();
  throw str;
}
```

Fig. 2. C++ custom type computation code for `==`

emitted checkers. The representation of terms is optimized by generating code for custom classes for each expression declared or defined in the signature. The parser generates instances of these classes when parsing. Binding expressions (λ- and Π-expressions) are parsed in such a way that each bound variable is represented as a distinct instance of a `DefExpr` class, with all uses of the variable represented as references to that same instance. In the C++ checkers, this is achieved using a trie rather than an STL `hash_map`, for performance reasons.

The Signature Compiler inlines the code needed to compute the type of an application of a constant declared or defined in the signature. The expected types of arguments are hard-coded into the emitted checkers, and the substitutions which must normally be performed at run-time to compute the return type of an application of a dependently typed function are performed instead during signature compilation. The emitted checkers thus completely avoid the expensive operation of substitution when computing the return type of an application of a constant declared or defined in the signature.

For example, the custom checker generated by sc produces the code shown in Figures 2 and 3 for cases for `==` and `impi` in a switch statement over all possible expressions. Note that since `==` cannot serve as a C++ or Java identifier, sc encodes this name using decimal ASCII character codes. Comments document the connection to the original name. The function `areEqualNuke` tests convertibility and additionally deletes the memory for the expressions it is given. Since the two subexpressions of any `==` expression must be terms (of type `trm`), the custom code for the `imp` case checks this condition. The type `o` is then returned. The code for `impi` is the result of substitution during signature compilation, and hence directly computes the appropriate substituted types.

The custom checker also has customized code for convertibility checking. For example, consider the case of expanding defined constants of functional type where they are applied. The exact expression resulting from substituting the arguments for the λ-bound variables is known from the signature, and thus code to build it directly is generated for the custom checker by Signature Compiler.

3 Benchmarks

Results on two families of benchmarks are reported in this section, using both explicit and implicit LF. The first are the EQ benchmarks, a family of proofs of statements of the form "if $f(a) = a$ then $f^n(a) = a$", for various sizes n. The proofs are structured (via deliberate inefficiency) to use both hypothetical and parametric

```
case /*impi=*/ Ximpi_EXPR: {
  /*impi=*/ XimpiExpr *e = (/*impi=*/ XimpiExpr *)_e;
  DefExpr *innervar1 =
          new DefExpr("na",new /*pf=*/ XpfExpr(e->e1),
                           new IdExpr("na"));
  if( (areEqualNuke(computeType(e->e1),
                    new /*o=*/ XoExpr()) &&
       areEqualNuke(computeType(e->e2 ),
                    new /*o=*/ XoExpr()) &&
       areEqualNuke(computeType(e->e3 ),
                    new PiExpr( innervar1,
                                new /*pf=*/ XpfExpr(e->e2))) ))
    return new /*pf=*/ XpfExpr(new /*imp=*/
                               XimpExpr(e->e1,e->e2));
  throw str;
}
```

Fig. 3. C++ custom type computation code for `impi`

reasoning, central aspects of the LF encoding methodology, as well as β-reduction and defined constants. The second are the QBF benchmarks. To obtain these, a simple Quantified Boolean Formula solver was written. This solver reads benchmarks in the standard QDIMACS format, and emits proof terms showing either that the formula evaluates to true or to false. Easy benchmark formulas, obtained from www.qbflib.org are solved to generate the proof terms.

Results on these two families of benchmarks are obtained using five checkers: the custom C++ and Java checkers generated by sc, Twelf, sc itself, and the flea checker [5]. Twelf version 1.5R1 is included as a widely used interpreting checker. The Signature Compiler itself implements an interpreting checker, using similar infrastructure as the custom checker. Comparing sc with the generated checker thus demonstrates the effect of the specializing optimizations. The flea checker is a highly tuned interpreting LF checker, which additionally implements context-dependent caching of computed types. Such caching is not implemented in sc or the emitted checkers. Note that the flea checker does not support implicit LF, infix directives (thus requiring prefix forms of the benchmarks), or printing of parsing times. These checkers are the only publicly available high-performance LF checkers the authors are aware of.

The results for the EQ benchmarks are shown in Figures 4 and 5, and for the QBF benchmarks in Figures 6 and 7. Parsing times, where available, are shown in parentheses. Experiments are averages of three runs on a 2GHz Pentium 4 with 1.5 GB main memory. The C++ and Java checkers emitted by sc were compiled with g++ and gcj, respectively, version 3.4.5. For the QBF benchmarks, a timeout of 30 minutes was imposed (on the toilet_02_01.2 benchmark, Twelf finished in just under that time on one run, so the average time for three runs is included). Note that the redundancy in the QBF explicit benchmarks explains flea's good performance.

4 Conclusion

The Signature Compiler is the first tool of its kind, supporting compilation of an LF signature to optimized C++ or Java backend checkers specialized for that signature. Results on two families of benchmarks, including one family of proofs of QBF benchmarks, show order-of-magnitude performance improvements for emit-

n	size	sc: C++	sc: Java	Twelf	sc (interp.)	flea
100	464 KB	0.2 (0.1)	1.2 (1.0)	4.1 (1.5)	2.0 (0.5)	0.8
150	1.01 MB	0.4 (0.2)	2.4 (2.1)	8.7 (2.6)	4.1 (1.1)	1.6
200	1.77 MB	0.6 (0.3)	4.1 (3.5)	16.2 (5.2)	7.1 (1.9)	2.7
250	2.74 MB	0.9 (0.5)	6.2 (5.4)	26.8 (9.2)	10.9 (3.0)	4.2
300	3.92 MB	1.2 (0.7)	8.8 (7.6)	39.7 (13.1)	15.5 (4.2)	6.0
350	5.30 MB	1.7 (1.0)	11.9 (10.4)	52.3 (16.1)	21.0 (5.7)	8.2

Fig. 4. Runtime for EQ benchmarks (in seconds), explicit form

n	size Twelf	size sc	sc: C++	sc: Java	Twelf
100	80 KB	87 KB	0.06 (0.03)	0.31 (0.26)	1.4 (0.2)
150	166 KB	176 KB	0.11 (0.05)	0.54 (0.45)	3.0 (0.4)
200	281 KB	295 KB	0.17 (0.08)	0.87 (0.68)	5.3 (1.0)
250	426 KB	444 KB	0.23 (0.11)	1.25 (1.03)	7.8 (1.9)
300	602 KB	623 KB	0.31 (0.14)	1.78 (1.34)	11.7 (2.2)
350	807 KB	833 KB	0.41 (0.18)	2.28 (1.80)	16.7 (2.5)

Fig. 5. Runtime for EQ benchmarks (in seconds), implicit form

name	size	sc: C++	sc: Java	Twelf	sc (interp.)	flea
cnt01e	2.2 MB	0.9 (0.6)	6.3 (5.8)	28.6 (7.0)	6.8 (2.2)	2.8
tree-exa2-10	2.7 MB	1.3 (0.8)	7.5 (6.9)	34.4 (8.5)	9.4 (2.8)	2.9
cnt01re	3.9 MB	1.7 (1.1)	10.7 (9.6)	56.7 (12.4)	12.3 (3.9)	5.1
toilet_02_01.2	9.7 MB	4.2 (2.7)	24.5 (22.0)	1809 (35.5)	30.6 (9.5)	10.5
1qbf-160cl.0	16.6 MB	6.4 (4.6)	41.3 (38.2)	timeout	44.5 (16.2)	14.6
tree-exa2-15	32.5 MB	15.9 (9.7)	86.1 (75.9)	timeout	114.1 (33.6)	25.8
toilet_02_01.3	96.4 MB	42.9 (27.8)	277.7 (241.2)	timeout	313.0 (99.0)	105.2

Fig. 6. Runtime on QBF benchmarks (in seconds), explicit form

name	size Twelf	size sc	sc: C++	sc: Java	Twelf
cnt01e	167 KB	184 KB	0.2 (0.1)	1.5 (1.4)	7.2 (0.6)
tree-exa2-10	345 KB	392 KB	0.4 (0.1)	2.1 (1.8)	8.9 (0.7)
cnt01re	250 KB	274 KB	0.3 (0.1)	1.9 (1.6)	12.3 (0.9)
toilet_02_01.2	0.9 MB	1.1 MB	1.0 (0.3)	4.1 (3.3)	38.0 (2.7)
1qbf-160cl.0	1.4 MB	1.5 MB	0.8 (0.4)	4.9 (4.6)	197.7 (4.5)
tree-exa2-15	3.9 MB	4.5 MB	4.7 (1.3)	14.4 (10.5)	timeout
toilet_02_01.3	7.6 MB	8.5 MB	9.4 (2.4)	28.1 (19.3)	timeout

Fig. 7. Runtime on QBF benchmarks (in seconds), implicit form

ted checkers over Twelf and sc itself, and substantial improvements over the flea checker. A form of implicit arguments is supported by sc, offering further space and performance improvements. Future work includes further support for proofs from decision procedures: the second author is proposing LF, backed by the Signature Compiler, as appropriate technology for a standard proof format for the SMT-LIB (Satisfiability Modulo Theories Library) initiative.

The authors wish to thank the anonymous reviewers for their comments on the paper.

References

[1] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[2] G. Necula. Proof-Carrying Code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, January 1997.

[3] G. Necula and P. Lee. Efficient representation and validation of proofs. In *13th Annual IEEE Symposium on Logic in Computer Science*, pages 93–104, 1998.

[4] F. Pfenning and Carsten Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *16th International Conference on Automated Deduction*, 1999.

[5] A. Stump and D. Dill. Faster Proof Checking in the Edinburgh Logical Framework. In *18th International Conference on Automated Deduction*, pages 392–407, 2002.

```
o : type.
trm : type.

== : trm -> trm -> o.
imp : o -> o -> o.
all : (trm -> o) -> o.
f: trm -> trm.

%infix left 3 ==.
%infix left 5 imp.

pf : o -> type.

refl : {x:trm} pf (x == x).
symm : {x:trm} {y:trm} pf (x == y) -> pf (y == x).
trans : {x:trm} {y:trm} {z:trm}
        pf (x == y) -> pf (y == z) -> pf (x == z).
congf : {x:trm} {y:trm} pf (x == y) -> pf ((f x) == (f y)).

mp : {p:o} {q:o} pf (p imp q) -> pf p -> pf q.
impi : {p:o} {q:o} (pf p -> pf q) -> pf (p imp q).

alli : {P:trm -> o} ({x:trm} pf (P x)) -> pf (all P).
alle : {P:trm -> o} {t:trm} pf (all P) -> pf (P t).

a : trm.
b : trm.
c : trm.

g : trm -> trm = [x:trm] f x.
```

Fig. A.1. LF signature for the EQ benchmarks

```
pol : type.
pos : pol.
neg : pol.

opp : pol -> pol -> type.
opp1 : opp pos neg.
opp2 : opp neg pos.

o : type.

conn : pol -> o -> o -> o.
not : o -> o.
quant : pol -> (o -> o) -> o.
bval : pol -> o.

Equiv : o -> o -> type.

%infix right 3 Equiv.

refl : {p:o} p Equiv p.
trans : {p:o}{q:o}{r:o} p Equiv q -> q Equiv r -> p Equiv r.

connc : {b:pol} {p1:o} {p2:o} {q1:o} {q2:o}
           p1 Equiv p2 -> q1 Equiv q2 -> conn b p1 q1 Equiv conn b p2 q2.
connz1 : {b:pol} {bb:pol} opp b bb ->
         {q:o} conn b (bval bb) q Equiv (bval bb).
connz2 : {b:pol} {bb:pol} opp b bb ->
         {q:o} conn b q (bval bb) Equiv (bval bb).
connu1 : {b:pol} {q:o} conn b (bval b) q Equiv q.
connu2 : {b:pol} {q:o} conn b q (bval b) Equiv q.

nott : not (bval pos) Equiv (bval neg).
notf : not (bval neg) Equiv (bval pos).

quantz : {b:pol}{bb:pol} opp b bb ->
         {a:pol}{p:o -> o} p (bval a) Equiv (bval bb) ->
         quant b p Equiv (bval bb).
quantu : {b:pol}{p:o -> o}
         p (bval pos) Equiv (bval b) ->
         p (bval neg) Equiv (bval b) ->
         quant b p Equiv (bval b).
quantn : {b:pol} {p1:o} quant b ([x:o]p1) Equiv p1.
quantc : {b:pol}{p1:o -> o}{p2:o -> o}
         ({x:o} (p1 x) Equiv (p2 x)) ->
         quant b p1 Equiv quant b p2.
```

Fig. A.2. LF signature for the QBF benchmarks