Compilation as Rewriting in Higher Order Logic

Guodong Li and Konrad Slind

School of Computing, University of Utah
{ligd, slind}@cs.utah.edu

Abstract.

be isolated clearly and specified as term rewrites, making it easy to construct a “new” certified compiler by applying the rewrites in a different order. The

2. *Prove Dynamically*. A per-run correctness check is performed. The result of a rewrite is verified each time it is applied to a program.

The format of a rewrite rule is [name] redex $\longleftrightarrow$ contractum $\square$ P. It specifies an expression that matches the redex can be replaced with the contractum provided that the side condition contain boolean constants and ; then apply rewrite dules based on the de Modgan theodems to moving negations in over the connectives (conjunction, disjunction and conditional expdessions). Meanwhile the decision pdocedure fod formulas of Presburger arithmetic is called to simplify and nodmalize arithmetic expdessions (this is essentially a proof-based implementation of *constant folding*).

[

In order to avoid unnecessary let-expression insertion in subsequent phases, during this transformation we rewrite an expression e to atom e, where atom $=$ $\lambda x.\, x$

the same names in the inlining function and inlined function, no problem will be incurred during substitution since the logical framework will capture program scope and rename variables automatically. For a recursive function, we avoid code explosion by expanding its body for only a certain number of times. The expression obtained from inline expansion is further simplified by applying other transformations such as the let-expression simplifications and constant folding until no more simplications can be made.

[fun_intro]	let $v = \lambda x.e_1[x]$ in $e_2[v] \;\longleftrightarrow\;$ let $v =$ fun $(\lambda x.e_1[x])$ in $e_2[v]$	
	$\square \quad$ size $e_1 < t$	
[unr6 _rec]	let $f =$ fun $e_1[f]$ in $e_2[f] \;\longleftrightarrow\;$ let $f =$ fun $(e_1[e_1[f]])$ in $e_2[f]$	
	$\square \quad$ size $e_1 < t$	
[inline_expand]	let $f =$ fun e_1 in $e_2[f] \;\longleftrightarrow\; e_2[e_1]$	

3.4 Closure Conversion

4 Code Generation

After the transformations in Section 3 are over, a source program has been converted into equivalent form that is much closer to assembly code. This form, with syntax shown in Fig.4, is called Functional Intermediate Language (FIL).

x ::= *r* | *m* | *i*	register variable, memory variable and integer
y ::= *r* | *i*	register variable and integer
v ::= *r* | *m*m	

Based on the basic rules, we derive some advanced rules for more complicated control flow structures such as conditional jumps, tail recursions and function calls:

$$
\dfrac{\dfrac{\dfrac{I_1 \;\; pre \;\; I_2 \qquad (w_2, w_1) \qquad\qquad I_2 \;\; S \;\; I_3 \qquad (f \;\, w_1, \; v_1)}{I_1 \;\; pre \;\; S \;\; I_3 \qquad (\text{let } w_1 = w_2 \text{ in } f \;\, w_1, v_1)}\;\text{seq} \qquad I_3 \;\; \text{post} \;\; I_4 \qquad (v_1, v_2)}{I_1 \;\; pre \;\; S \;\; post \;\; I_4 \qquad (\text{let } v_1 = (\text{let } w_1 = w_2 \text{ in } f \;\, w_1) \text{ in } v_1, \; v_2)}\;\text{seq}}{I_1 \;\; pre \;\; S \;\; post \;\; I_4 \qquad (f \;\, w_2, v_2)}\;\text{let\_def}
$$

6. John Hannan and Frank Pfenning, *Compiler verification in LF*, Proceedings of the