Compiler Construction in Higher Order Logic Programming

Chuck C. Liang

Department of Computer Science
Hofstra University
Hempstead, New York 11550
email: cscccl@hofstra.edu

Abstract. This paper describes a general method of compiler implementation using *higher order abstract syntax* and logic programming. A working compiler written in λProlog is used to demonstrate this method. Various stages of compilation are formulated as higher order logic programming including parsing and the generation of higher order representations, type checking, intermediate representation in continuation-passing style, and machine-dependent code generation. The performance overhead of using higher order representations is also addressed.

1 Introduction

The construction of compilers and translators for programming languages involves much computation that is of a symbolic and deductive nature. Although there has been much study of these aspects of program analysis in the context of declarative languages, logic programming is still not used as a general tool in compiler construction. Implementations of compilers still rely principally on imperative languages, which are less capable of exploiting the inherently declarative aspects of compilation. In particular, *higher order abstract syntax* [27, 38], a technique that uses λ-terms in the representation of programs, has been shown to be a valuable tool in the declarative analysis and transformation of programs. This new form of abstract syntax however, is still not applied to practical compiler construction. Studies of higher-order abstract syntax have so far been limited to very small examples.

The aim of the project described here is to develop a method of compiler construction using higher order abstract syntax in general, and a higher order logic programming language in particular. Some of the formulations described here are syntheses and extensions of previous works, while new methods are also introduced when needed. We shall address the various stages of compiler construction, from parsing to machine-dependent code generation, in the context of describing an experimental compiler written in λProlog. Due to space constraints, the discussion here is necessarily limited to outline form, and examples are small to provide clarity. The full implementation, larger examples, and additional notes can be found at `www.cs.hofstra.edu/~cscccl/hocompiler/`.

It should be stated that we can only present and demonstrate a selected set of compilation techniques. This paper does not seek to be comprehensive in addressing the massive number of approaches developed for compilation, both in general and for particular styles of programming languages. It is inevitable that readers will inquire as to how our body of techniques will be applicable in specific settings, as for instance, in abstract interpretation of logic programs, polymorphism and inheritance in object oriented languages, or register allocation for particular machine architectures. We cannot hope to answer all such questions here. Our purpose is to conduct an experiment on the applicability of higher order abstract syntax in realistic compilation. We hope to *open the door* to the investigation and incorporation of other techniques in this declarative setting.

2 Higher Order Abstract Syntax and λProlog

Conventional abstract syntax representations of source programs cannot capture the *locality* of names that is essential in high level programming languages. As a result, compilers have traditionally relied on external structures, namely symbol tables, to hold this information. Higher order abstract syntax (HOAS) uses λ-terms to represent source programs, thereby capturing the locality of names in a natural and declarative manner. The application of HOAS requires a language in which λ-terms can be used as data-structures, *and* in which enough constructs exists for reasoning with λ-terms. Functional languages may have functions in data structures but they cannot, for example, recognize α-equivalent terms. Thus even languages such as ML cannot fully exploit a higher order representation. The logic programming language λProlog [31] extends standard Prolog by using λ-terms and higher-order unification [18] in place of first order terms and unification. It allows for universal quantification and implication in goals. The following are typical λProlog clauses:[1]

```
kind tm, ty type.                % object-level terms and types
type app tm -> tm -> tm.         % application
type abs (tm -> tm) -> tm.       % abstraction
type --> ty -> ty -> ty.         % type constructor
infixr --> 10.
type typecheck tm -> ty -> o.    % typing judgment

typecheck (app A B) T :- typecheck A (S --> T), typecheck B S.
typecheck (abs A) (S --> T) :-
    pi v\ (typecheck v S => typecheck (A v) T).
```

The `kind` and `type` clauses declare the signature of an object-level language, which in this case is that of typed λ-terms. Object-level terms and types are categorized by `tm` and `ty` respectively at the meta-level. The infix operator `-->` is

[1] These clauses are actually $L\lambda$ clauses. $L\lambda$ [25] is a sublanguage of λProlog that uses a decidable and deterministic form of higher order unification. Most λProlog clauses in practice are in fact in $L\lambda$.

the object-level functional type constructor, which should not be confused with the λProlog type constructor `->`. The predicate `typecheck` represents type judgment. Application is written in Curried form: `(f x)` instead of `f(x)`. The first `typecheck` clause is an ordinary first order Horn clause. In the second clause, `A` is a λ-term variable. λ-abstraction is written with the infix symbol \ (e.g., the identity is `x\x`). Implication '`=>`' and universal quantification '`pi v\`' can appear in the bodies of clauses. The operational meaning of these new constructs are derived from intuitionistic proof theory: $\forall x.A$ holds if A holds under a fresh *eigenvariable* x not appearing free elsewhere. $A \Rightarrow B$ holds if B holds with definite clause A locally visible. When combined with higher-order unification (or $L\lambda$ unification [25]), these constructs allow recursion over the structure of λ-terms while preserving the integrity of bound variables. They permit a language to analyze and manipulate the *internal structure* of abstractions, making it suitable for applying HOAS. The two λProlog clauses above implement type checking for object-level λ-terms, and are representative of the kind of clauses used in our compiler.

It should be noted that our use of λProlog is secondary to our use of higher order abstract syntax. Future developments may permit the application of HOAS in other systems, possibly including functional-logic hybrid languages. Previous investigation [28] has determined the critical characteristics of any language that can truly support HOAS (illustrated by the λProlog extensions described above). But experimental compiler implementation also requires the practical features of a realistic programming language. λProlog is currently one of the only languages that satisfy both criteria. One significant alternative is Elf [37], which has been used for the theoretical investigation of many topics pertaining to programming languages, including compiler verification [15]. The justification of our preference for λProlog lies primarily in the availability of a high-performance implementation, which is discussed in Sections 3 and 9.

For more information on λProlog and higher order abstract syntax, the reader is asked to consult [27, 31].

3 Obstacles to Application

From the beginning, work in higher order abstract syntax have pointed in the direction of its application in compiler writing. It has been known for some time, for example, that HOAS can give declarative formulations of type checking. Other works include using HOAS for the recognition of tail recursion [32], and various program transformations [14]. Interpreters for very small languages (without parsing) have also been formulated. These experiments are useful in demonstrating (and even proving) theoretical aspects of programming languages. However, they are uniformly limited in scale. None has been applied to actual compilation.

We can identify two major factors that have discouraged the application of HOAS as an actual compilation paradigm:

1. Lack of a high-performance implementation of a suitable programming language, one that can support HOAS.

2. Lack of general tools for *deterministic parsing* (such as Yacc), which are needed to generate HOAS representations from source text.

Recent developments have significantly improved the situation with both of these obstacles. First of all, work in improving the representation and normalization of λ-terms, such those using DeBruijn indices and explicit substitutions [1, 30], have lessened the overhead of systems that need to reason with λ-terms. The recent *Teyjus* implementation of λProlog [33] uses such advanced techniques. Teyjus is also *compiler*-based in that it extends Warren's Abstract Machine to accommodate λProlog. In addition, Teyjus has a module system and sufficient I/O capabilities to support compiler construction. We shall further discuss how Teyjus lessens the overhead of HOAS in Section 9. The second obstacle listed above is addressed in the next section.

4 Parsing and the Generation of λ-Syntax Trees

Realistic compilation with HOAS cannot begin unless higher-order representations can be constructed automatically from source text. A parser, coupled with sufficiently expressive "semantic actions" can be used for this purpose. But standard parsing tools (Yacc and derivatives) have been inaccessible to logic programming languages that can support HOAS. Parsing in logic programming has traditionally relied on Definite Clause Grammars (DCG - [35]). Though general in scope, DCGs have no inherent capability to resolve non-determinism. For example, there is nothing that would prevent the user from writing an ambiguous grammar as a DCG. Compilers using DCGs do exist (see [4] for a survey of early work). However, the programmer will have to *manually* provide the mechanisms needed for deterministic parsing (such as lookahead symbols). In contrast, Yacc-style tools generate deterministic parsers from appropriate grammars automatically. Even if DCGs are a viable solution for general compiler writing, they must still be extended to the higher order case, with richer constructs. Attempts to thus extend DCGs exist [8, 19], but they have not led to any usable tools.

A deterministic, bottom-up parser generator has recently been developed for λProlog, and is described in [23]. The parser generator requires a different restriction of LR grammars than the LALR grammars used in other Yacc-style tools. It requires the user to write a few more productions: the grammar used in our compiler required eight more productions (out of approximately seventy) compared to traditional Yacc. However, much of the character of Yacc-style parser construction is preserved. The essence of this parser generator is illustrated by the following pair of grammars:

LR:	$S \to A$	BRC:	$S \to A$
	$A \to (B) \mid x$		$A \to (B) \mid x$
	$B \to (A) \mid x$		$B \to CA) \mid x$
			$C \to ($

Both grammars generate the same language, and both grammars are sensitive to whether the terminal symbol x is surrounded by an even or odd number of pairs of parentheses. For the LR grammar on the left this distinction requires the generation of a finite state machine, which is used to resolve the "reduce-reduce" conflict between $A \to x$ (even) and $B \to x$ (odd). The *bounded right context* (BRC) grammar[2] on the right is also an LR grammar. However, the *state information* as to whether an even or odd number of '('s have been read is maintained by the grammar itself, in the form of the extra non-terminal symbol C. In a bottom-up derivation using the second grammar, new '(' symbols will be alternately shifted on to the stack (if it's an odd occurrence) and reduced to C (if it's an even occurrence). x must be reduced to A if it's preceeded by C on the stack, and to B if preceeded by a '('. There is no need to maintain state information externally. Every LR grammar has an equivalent BRC grammar [12, 20]. In the context of logic programming, the advantage of this simplification of LR parsing is that it facilitates the formulation of *deterministic bottom-up parsing* as *deterministic proof search.* Instead of implementing general LR parsing algorithms by brute force, a process that destroys much of the character of logic programming, our parser generator constructs inference rules such as the following (which should be read bottom-up):

$$\frac{\ldots(B}{\ldots(x}\ \mathit{Reduce} \qquad \frac{\ldots CA}{\ldots Cx}\ \mathit{Reduce} \qquad \frac{\ldots(C}{\ldots((}\ \mathit{Reduce} \qquad \frac{\ldots C(\sigma}{\ldots C(}\ \mathit{Shift}$$

(σ is the next input symbol.) A pairwise matching of the bottom sides of these rules will confirm that there are no "reduce-reduce" or "shift-reduce" conflicts[3]

The parser generator also contains a customizable lexical analyzer. Since the parser generator is written in λProlog and outputs λProlog code, the semantic actions associated with grammar rules are in fact λProlog goals. These goals can be written so that they generate λ-term syntax trees as source text is parsed bottom-up. The formation of abstractions over text is accomplished by the following *copy* clauses, which first appeared in [26]:

```
copy (app A B) (app C D) :- copy A C, copy B D.
copy (abs A) (abs B) :- pi v\ (copy v v => copy (A v) (B v)).
```

These clauses formulate substitution over object-level λ-terms. The goal

```
pi u\ (copy u S => copy (A u) B)
```

succeeds only if `B` is β-equivalent to `(A S)`. For example, given source text (in pseudo-ML syntax)

```
"fun f x = (g x);"
```

[2] BRC grammars were well-studied in the 1960's and 70's (e.g. see [11, 12, 16]), but they were eventually overshadowed by general LR parsing. However, they were never considered in the context of logic programming.

[3] The connection between parsing and formal deduction has been studied in the past [36], but *deterministic* parsing in such a framework has received little attention.

the subexpression `"(g x)"` will be parsed first (since it's bottom-up) into an ordinary syntax tree, namely `(app "g" "x")`. The abstraction over `"x"` will be seen next, and at this point the λ-term syntax tree

```
(abs v\ (app "g" v))
```

will be generated by the goal

```
pi u\ (copy u "x" => copy (A u) (app "g" "x")), !
```

The meta-level variable `A` will be instantiated to the meta-level abstraction `v\(app "g" v)`. Any subsequent `"x"` encountered in the source text cannot be confused with the bound variable in this abstraction[4]. We use the standard logic programming control feature "`!`" when necessary. Although "logically perfect" alternatives can be be fashioned with some effort, we are also concerned with efficiency. Our use of λProlog is as a *real programming language.* The declarative paradigm at work here is not merely logic programming, but also the higher-order form of abstract syntax. An alternative method of creating the abstraction `A` is to use λProlog's built-in higher order unification procedure and provide the programmer means to bias *projections* over *imitations.* The unification goal

```
(A "x") = (app "g" "x")
```

will then accomplish the same purpose.

5 An Experimental Compiler

Once an adequate means of generating higher order representations from source text is available, we can finally begin to apply HOAS in formulating compilers. This and the next few sections describe how HOAS is used in each of the basic compilation stages.

The source language we have chosen for our experimental compiler is an imperative-style language with enough flexibility for experimenting with new features. Many aspects of the language, including its syntax, are based on the "Tiger" language described in [3], which has versions in ML, Java, and C. It is hoped that this choice will facilitate the comparison between our methods of compiler writing with those of other styles of programming. The target language is Sparc assembly language. An existing assembler (gcc in its role as an assembler) is used in the last step to produce Sparc executables. Except for this last step, every aspect of the compiler is written in λProlog.

Although the scale of the source language and compiler is still limited, it is a significant expansion of previous experiments that used HOAS for implementing programming languages (e.g. [14, 32]). These early experiments typically used

[4] The clauses presented here are stylized for brevity. In reality different copy causes must be defined for each type, i.e, for expressions and for strings. A default clause that copies terms to themselves is also needed so that `"g"` is preserved before its abstraction is encountered.

source languages with only four or five constructs. None of them involved the building of a complete system that includes efficient parsing, intermediate language, and realistic target language. Clearly a new level of experimentation is required if higher order abstract syntax is to acquire more than theoretical importance.

Our source language admits nested scopes and functions within functions, but it is not higher ordered. To partially compensate for this lack, we implemented Java-style classes and objects (without inheritance, so far)[5]. This means that a function requires an additional parameter that points to heap records with bindings for its free variables (the Sparc registers `%o5` and `%i5` are reserved for this purpose). Some of the flavor of computing with closures is therefore captured. A higher order, functional language will require a more general treatment of closures. The runtime stack cannot be used easily since local variables may persist in closures, necessitating a garbage collector. These are topics that we wish to explore in the future, but we decided to first demonstrate our methods on a conventional source language.

λ-Syntax Trees

HOAS is applicable to more than one stage of compiler writing (see Section 7). We shall refer to the specific structures generated by the parser as λ-*syntax trees*. We shall continue to use "higher order abstract syntax" to refer to the general technique of using λ-terms in representations.

The λProlog signature for λ-syntax trees of the source language is similar to the abstract syntax structures described in [3], except of course, for its higher-order components. The constructs of the language are defined by type declarations such as the following:[6]

```
kind texp type.   % meta-level type for values
kind ttype type.  % type for object-level types
kind tdec type.   % type for declarations
type intexp int -> texp.     % integer constant
type strexp string -> texp.  % string constant
type opexp string -> texp -> texp -> texp. % binary op.
type ifexp texp -> texp -> texp -> texp.  % if
type whileexp texp -> texp -> texp.       % while
type callexp texp -> (list texp) -> texp.  % function call
type letexp tdec -> texp -> texp.     % let
type dabs (tvar -> tdec) -> tdec. % declaration abstraction
type pabs (tvar -> texp) -> texp. % parameter abstraction
% etc ...
```

[5] Classes and instances are represented with bound variables, but (currently) their members are not.

[6] The exact definition of our source language, as well as its abstract syntax structure, is transitory as we explore different strategies for compilation.

The meta-level type for λ-syntax tree expressions is `texp`. The most important higher-order constructors are `dabs` and `pabs`, which abstracts over declarations and formal parameters of functions respectively. `dabs` abstractions are used in `letexp` statements to abstract over the names of a set of (possibly mutually recursive) identifier declarations.

6 Type Checking and Type Inference

A number of compilation strategies, such as static analysis and partial evaluation, can potentially benefit from the ability to reason with a higher order representation directly. Some of these subjects have already been studied in the context of HOAS.

In this section we focus on one essential component of most compilers: type checking. The two type checking clauses presented in Section 2 do not completely suffice in realistic settings. One feature we can expect to find in many languages is mutually-recursive definitions. Higher order logic programming offers interesting solutions for type checking and inference under mutual recursion. Consider an ordinary piece of source code such as

```
let
    var a := 10
    function g(n:int) = if n>0 then f(n-1)
    function f(n) = if n>0 then g(n-1)
in  g(a)  end
```

The type of function `f` is not known while type checking `g`. We also may not have the benefit of any "headers," (as is the case with function `f`). In the λ-syntax tree, this let-structure is represented by two abstractions over a pair consisting of first the declarations followed by the body of the let. In the sample code below, `typeof` is a predicate that associates an expression with a type. `dabs` is a second-order constructor that represents abstraction over a list of declarations followed by an expression in which they appear (its argument will be a λ term). `vardec` and `fixptdec` are constructors for variable and functions declarations respectively. `intexp` and `callexp` are constructors for integer constants and function calls respectively. `namety` is a constructor for (object-level) primitive types, and `-->` is an infix constructor for (object-level) functional types.

```
typeof (let (dabs T)) U :-
  pi v\ (typeof v A => typeof (T v) U).
typeof (vardec Var Exp) :- typeof Var A, typeof Exp A.

typeof (fixptdec F f\(pabs (T f))) (A --> B) :-
     typeof F (A -> B),
     pi f\ (typeof f (A --> B) =>
        pi x\ (typeof x A => typeof (T f x) B)).
```

```
typeof (callexp Fun Arg) B :- typeof Fun (A --> B), typeof Arg A.
typeof (intexp N) (namety "int").
% etc ...
```

The first clause is key: for each abstraction it introduces an eigenvariable for the name of the identifier that will be declared. It also introduces a (meta-level) *free variable* for the initial type of every identifier. Thus for the sample program above the types for `f` and `g` are assumed to be distinct free variables initially. These free variables will be instantiated (via unification) into appropriate type expressions when the declarations are processed. For example, for a variable declaration `vardec`, the type associated with the identifier `Var` must match the type of its initial value `Exp`. Subsequent occurrences of `Var` are therefore constrained to have types consistent with the type of `Exp`.

Unification is a natural tool for type checking that imperative languages lack. A conventional compiler will also require a symbol table to keep track of types. In the higher order logic programming formulation the symbol table is replaced by the declarative construct of intuitionistic implication (`=>`), which, together with λ-abstraction, respects scopes in a natural way.

The above formulation of typing suffices for a non-polymorphic language (as is the case with the language used in our experimental compiler). A language with generic type variables such as ML would require the use of a higher order representation of type *schemes* [5]. A quantified type such as $\Pi a.a \rightarrow a$ has a natural higher order representation where the type variable a is bound by an abstraction, i.e., $\Pi \lambda a.a \rightarrow a$ where Π is now a second order constructor. A principal type scheme can now be thought of as an α-equivalence class of terms. Type inference using such representations of type schemes has also been formulated in HOAS [13, 22], though they require an indirect way of using meta-level unification than the "mono-typing" described above. In [22] a type inference algorithm is described that targets ML-style let-polymorphism. The expression

```
let val f = (fn x => x) in (f f) end;
```

is typed by associating `f` with the type scheme $\Pi \lambda a.a \rightarrow a$. α-equivalent instances of this type scheme are then associated for each occurrence of `f` in the body of the let-expression: $\Pi \lambda a.a \rightarrow a$ and $\Pi \lambda b.b \rightarrow b$. Since the type variables in these instances are bound at the meta-level, they cannot be confused. In a logic programming implementation, each instance can be instantiated with a distinct meta-variable, allowing the correct derivation of the type of the let-expression.

7 A Continuation Passing Style Representation

Modern compilers require intermediate languages. One such language that is based on the λ-calculus already exists, and there is considerable work on translating programs to *continuation passing style* ("CPS" - see for example [2, 9, 21, 39, 40]). What can higher order abstract syntax and higher order logic programming add to this well-studied and well-applied subject? In a language that cannot fully support HOAS, including functional languages, a meta-level CPS representation is, ironically, not higher ordered. For example, in [2] a CPS representation is defined in ML, but the ML data structure for CPS expressions is in fact first order. What should be bound variables are represented with ordinary symbols, and not with *abstractions in the meta language.* That is, CPS continuation functions are not represented using ML's native abstractions (`fn =>`). This is due to the inability of ML to *look inside* its abstractions, which is necessary during the analysis and transformation of CPS terms. A HOAS representation of CPS should be able to take advantage of the fact that a CPS term is after all, a λ-term.

We use a higher order CPS-*like* representation as the intermediate language in our experimental compiler. We differ from the traditional CPS representation in the following way. Instead of adding an extra continuation parameter to each function and operator, we modify the application of each function and operator so that they carry a *continuation abstraction.* Essentially, this form of CPS is equivalent to what is referred to as the *direct style* [6] or *A-normal form* [10] representation. It has been shown (see [10]) that this form of CPS is equivalent to the traditional form, with less clutter.

Our CPS intermediate language is essentially a miniaturized version of the source language. Each construct is modified to carry one or more continuations. It has the essential characteristics of CPS in that each operator and function is applied only to primitives. However, it preserves some of the top-down structure of a higher-level abstract syntax representation by duplicating certain essential constructs[7]. In the following signature of our CPS language, recall that `texp` is the type for expressions in the λ-syntax tree:

```
kind kexp type.   % meta-level type for continuation expressions
type cps texp -> kexp -> kexp.     % primitive operation
type kabs (texp -> kexp) -> kexp.  % continuation abstraction
type kreturn  texp -> kexp.        % return value, (stop)
type klet (texp -> kexp) -> kexp. % ‘‘let’’
type kformal (texp -> kexp) -> kexp.  % formal parameter
type karg texp -> int -> kexp -> kexp.  % actual parameter
type kcall texp -> int -> kexp -> kexp. % function call
type kvar tvar -> ttype -> texp -> kexp -> kexp. % variable
type kfunc tvar -> ttype -> kexp -> kexp.  % function
type klass tvar -> (list texp) -> kexp -> kexp. % class
type kif kexp -> kexp -> kexp -> kexp -> kexp.  % if-else
type kwhile kexp -> kexp -> kexp -> kexp.       % while loop
```

`kabs` is a second-order constructor for *continuation abstractions,* which are carried by all structures save the terminating `kreturn`. The constructs `kvar`, `kfunc`

[7] A strict adherence to the CPS principle can be excessive. For example, if we sequentialize the boolean condition of a while-loop before transforming its body, it will become unclear as to where the while loop actually starts. Some *top-down* structure can be helpful when generating code from the CPS representation.

and `klass` respectively define variables, functions and classes for their continuation. These constructs also carry limited type information due to the presence of classes. Although many constructs carry more than one continuation, only one of these is the "true" continuation surrounding the construct. For example, in `kif`, the condition, "then" and "else" parts of the construct are represented by continuations that terminate in `(kabs v\ kreturn v)`, or essentially $\lambda x.x$. Only the third `kexp` subexpression is the continuation that follows the if-else construct.

An expression such as $a + b * 2$ is represented by the `kexp` term

```
cps (opexp "*" (varexp (simplevar b)) (intexp 2))
    (kabs u\ (cps (opexp "+" (varexp (simplevar a)) u) K))
```

Here `K` is the continuation that always surrounds any expression. It should be emphasized that the symbols `a` and `b` are not strings but *bound variables*, since the higher-level λ-syntax tree is also higher-ordered.

The following are sample clauses that make up the (call-by-value) translation from the λ-syntax tree to the CPS representation. These clauses make use of decidable "higher order pattern" unification:

```
type atomic, primitive  texp -> o.
type formcps kexp -> texp -> kexp -> o.
atomic (intexp A).
atomic (varexp (simplevar S)).
primitive A :- atomic A, !.
primitive (opexp S A B) :- atomic A, atomic B.

formcps K A (cps A K) :- primitive A.
formcps K (opexp OP A B) AK :- atomic B, !,
  pi v\ (atomic v => (formcps K (opexp OP v B) (L v),
                      formcps (kabs L) A AK)).
formcps K (opexp OP A B) BK :-
  pi v\ (atomic v => (formcps K (opexp OP A v) (L1 v),
                      formcps (kabs L1) B BK)).
% etc ...
```

The `formcps` predicate takes a surrounding continuation (`K`), an expression, and the resulting continuation as parameters. The use of `!` here is for efficiency. It is worthwhile to note that *if* our source language consisted only of application and abstraction, then the CPS transformation would be defined as follows (where `kapply` and `kformal` represent general application and abstraction respectively):

```
type kapply texp -> texp -> kexp -> kexp.
formcps K (app A B) BK :-
  (pi u\ ( formcps (kabs v\ (kapply v u K)) A (L u)) ),
  formcps (kabs L) B BK.
formcps (kabs v\ kreturn v) (abs A) (kformal AK) :-
  (pi u\ (atomic u => formcps (kabs v\ kreturn v) (A u) (AK u))).
```

A similar higher order CPS transformation was given in [7], and was implemented in Elf. However, the purpose of this transformation was to prove a theoretical result with Elf, and was confined to application and abstraction.

It is also possible to use the source language itself to write CPS terms ("direct style"), but we decided to construct a new language to avoid some of the clutter of the source language.

A higher-order representation of CPS expressions can facilitate the formulation of various transformations on the intermediate form. For example, the following optimization clause pushes a purely functional operation inside the "else" case of an kif-statement, if the continuation abstraction (over `u`) is vacuous over the other components of the construct.

```
optimize (cps A (kabs u\ (kif C THEN (ELSE u) K)))
     (kif C THEN (cps A (kabs ELSE)) K) :- not (side_effect A).
```

The scopes of the variables in upper-case, as per Prolog convention, are over the entire clause, and therefore cannot contain free occurrences of `u`. `ELSE` must therefore abstract over `u`, allowing the continuation abstraction `kabs` to be pushed inside the else-case. We can similarly formulate dead variable elimination as vacuous abstractions, which are represented by the "higher order pattern" `x\A`. A first-order meta-level CPS formulation, without the benefit of higher-order pattern matching, would have to manually keep track of and possibly rename bound variables during such transformations.

It may be interesting to consider the formulation of intermediate languages other than CPS in HOAS. However, as argued in [2], there is *nothing wrong* with using CPS. The CPS representation is flexible enough to still allow the application of other techniques, even if they cannot take advantage of the higher order nature of CPS.

8 Code Generation

Higher-order abstract syntax is most useful in the *middle* stages of compilation. At the code generation stage, its benefits become less clear. However, there is nothing to indicate that it is necessarily an impediment if there are enough language features to support it.

Our experimental compiler generates machine-dependent assembly language text for the Sparc architecture [34]. We make use of Sparc's *register windowing* mechanism. Future work will involve an additional intermediate stage that will generate code for a general abstract machine.

The Sparc architecture is first represented in abstract syntax. A meta-level type `store` is defined to represent Sparc registers, memory locations, immediate constants, and labels. The meta-level type `inst` represents Sparc instructions. The principal predicate used for generating code for CPS expressions is

```
type gencode int -> int -> kexp -> store -> (list inst) -> o.
```

The first two arguments to this predicate keep track of an integer *label counter,* which is used to generate unique string labels. The "output" of this predicate is first of all a `store` location for the return value of a code fragment, followed by a list of instructions. On processing a CPS term such as a subtraction operation `(cps (opexp "-" A B) (kabs K))`, the appropriate instruction is generated for the subtraction. This involves finding a `store` location to hold the result of the operation. The `store` location, commonly a register, is then *passed* to the continuation abstraction. Each value (`texp`) expression (including `A` and `B` above) is therefore always associated with a store location. A predicate `assoc` is used to "remember" the store associated with each expression. A typical `gencode` clause has the form

```
gencode LC0 LC1 (cps A (kabs K)) RETURN [INST | INSTS] :-
  % generate expression A into instruction INST,
  pi v\ (assoc v STORE => gencode LC0 LC1 (K v) RETURN INSTS).
```

where `STORE` is the store location associated with the result of the instruction.

The principal difficulty in defining the `gencode` predicate is register assignment. No matter how declarative a meta-language is, it must deal with the fact that registers in a real machine must be used in special ways. Registers assigned to expressions need to be eventually deallocated, which requires `gencode` to keep track of state information. Our experimental compiler uses intuitionistic implication (`=>`) combined with `!` to model mutable information. This method is not entirely satisfactory, not only because it relies heavily on extra-logical features, but also because it is not efficient. An alternative would be to use a table-like data structure, something we have avoided up to this point. That is, `gencode` can carry extra parameters that *lists* expressions and their associated stores.

Perhaps a limited *linear logic programming language* [17] will be able to declaratively formulate register assignment. However, no practical, *higher-order* version of such a language currently exists.

9 Performance Overhead

The use of λ-terms in the representation of programs incurs an extra cost in speed and memory usage during compilation. With a naive implementation of λ-terms and λ-normalization, this extra cost can indeed be overwhelming. However, advanced implementation techniques are being developed [30] to reduce this overhead to a reasonable level.

Reasoning with λ-terms inevitably involves β- and/or α-conversion. For example, consider a CPS term of the form

```
cps A (kabs v\ (cps B (kabs u\ (cps C (kabs w\ ....
```

that involves n nested `cps` terms with `kabs` continuation abstractions. In order to process such a term we would need to look *inside* the λ-abstractions. This is accomplished (as described in the foregoing) by clauses of the representative form

```
process (cps A K) :- process A, process K.
process (kabs K) :- pi v\ (... => process (K v)).
```

The "`...`" represents clauses containing information associated with the eigenvariable `v`, such as the register it has been assigned. In a naive implementation of λ-term normalization, The β-reduction of `(K v)` would involve a complete traversal over the structure of `K`. All n `cps` subterms are thus *visited* after reducing just the first redex encountered. The reduced form of `(K v)` still has $n-1$ `cps` subterms, each with a similar continuation abstraction. The `process` goal will therefore be repeated for the abstraction at the next level, resulting in $n-1$ additional visitations of `cps` subterms, and so on. Altogether, the "processing" of the above term will involve $(n^2+n)/2$ *visitations* of its `cps` subterms, even though the CPS structure is naturally linear. Since the number of nested continuations n is proportional to the length of the source program, a naive β-reduction strategy can indeed be costly.

The redundant term traversals can be avoided using a more advanced implementation of β-reduction. In particular, an *explicit substitution* calculus [30] can be used to delay the traversal over the term structure until all β-redices have been encountered. The β-reduction of a term $(\lambda x.s)a$ can be *held in suspension* in a structure of the form $\{s;(a,x)\}$.

When a term such as this is encountered in another redex, we can *expand* the suspension environment to capture the substitution of more than one variable: $\{s';(a,x),(b,y)\}$. The final normalization of the term will involve a single traversal, substituting for all variables simultaneously. This refinement means that the CPS structure above can be "processed" in one pass, which is what one would expect with a conventional, first order representation. Such refinements are used in the Teyjus λProlog implementation [29].

A naive beta-reduction strategy may also need to replicate terms during substitution, resulting in greater memory usage. It was found that, with a modification to Teyjus so that it uses a naive reduction strategy, a heap overflow error occurred while compiling a program just a hundred lines long, while we've successfully compiled programs over a thousand lines long with (standard) Teyjus. Timing comparisons also support our above analysis: using explicit substitutions to delay traversal over terms results in an order-of-magnitude improvement in performance over a naive strategy. In one experiment, a one-page program was compiled (on a 1.1ghz workstation) in approximately 0.3 seconds with (standard) Teyjus and approximately 3.7 seconds with a naive reduction strategy. Future systems that can be used for the practical application of HOAS must implement enhanced strategies for representing and reducing λ-terms[8].

Work is continuing to further reduce the overhead of HOAS. One promising angle of approach is to optimize cases that appear most often in programs, namely the $L\lambda$ subset of λProlog with "β_0-reduction" and second order "pattern" unification [25].

---

[8] A forthcoming paper [24] will further discuss the relative advantages of various λ-term representation and reduction strategies.

10 Conclusion

Although our presentation has been in outline form, we hope to have conveyed a sense of the role that higher order abstract syntax can play during the principal stages of compiler construction. Certainly during portions of this process, namely the initial and final stages, HOAS plays little if any role. But for many of the operations and transformations performed on intermediate forms, HOAS can offer a simpler and cleaner formulation. Our experimental compiler, together with the continuing development of high-performance language support for λ-term representations, shows that HOAS can become useful for general compiler construction. Our compiler also provides a platform for studying new problems previously not considered. For example, with the joining of HOAS and CPS, it is possible to study whether certain optimizations, such as tail-recursion elimination, are better formulated as transformations on λ-syntax trees or on CPS terms.

From a general perspective, with the experimental compiler we have hopefully taken work on higher order abstract syntax to the next logical level. It is a necessary step in the development of higher-order abstract syntax into a practical, declarative programming paradigm. Clearly a great amount of work remain. We have immediate plans, for example, for expanding the compiler by implementing data-flow analyses via abstract interpretation. We now have a platform on which to continue such work.

Acknowledgments

The author wishes to thank Dale Miller and Gopalan Nadathur for their helpful comments.

References

1. M. Abadi, L. Cardelli, P. Curien, and J. Levy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.
2. A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.
3. A. W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.
4. J. Cohen and T. Hickey. Parsing and compiling using Prolog. *ACM Transactions on Programming Languages and Systems*, 9(2):125–163, 1987.
5. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Ninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–212, January 1982.
6. O. Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.
7. O. Danvy, B. Dzafic, and F. Pfenning. On proving syntactic properties of CPS programs. In *Proceedings of the Third International Workshop on Higher Order Operational Techniques in Semantics*, September 1999.

8. A. Felty. Defining object-level parsers in λProlog. In *Proceedings of the Workshop on the λProlog Programming Language*, 1992. Department of Computer and Information Science, University of Pennsylvania, Technical Report MS-CIS-92-86.
9. M. Fischer. Lambda calculus schemata. In *ACM Conference on Proving Assertions about Programs, SIGPLAN Notices 7*, number 1, pages 104–109, 1972.
10. C. Flanagan, A. Sabry, B. Duba, and M. Felleisen. The essence of compiling with continuations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 237–247. ACM Press, 1993.
11. R. Floyd. Bounded context syntactic analysis. *Communications of the ACM*, 7(2):62–67, 1964.
12. S. L. Graham. On bounded right context languages and grammars. *SIAM Journal on Computing*, 3(3):224–254, 1974.
13. J. Hannan. *Investigating a Proof-Theoretic Meta-Language for Functional Programs.* PhD thesis, University of Pennsylvania, August 1990.
14. J. Hannan and D. Miller. Uses of higher-order unification for implementing program transformers. In *Fifth International Logic Programming Conference*, pages 942–959, Seattle, Washington, August 1988. MIT Press.
15. J. Hannan and F. Pfenning. Compiler verification in LF. In *Seventh Annual IEEE Symposium on Logic in Computer Science*, Santa Cruz, California, June 1992. IEEE Computer Society Press.
16. M. Harrison and I. Havel. On the parsing of deterministic languages. *Journal of the ACM*, 21(4):525–548, 1974.
17. J. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.
18. Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.
19. S. Le Huitouze, P. Louvet, and O. Ridoux. Logic grammars and λProlog. In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 64–79. MIT Press, 1993.
20. D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.
21. D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. In *SIGPLAN 1986 Symposium on Compiler Construction, SIGPLAN Notices 21*, number 7, pages 219–233, 1986.
22. C. Liang. Let-polymorphism and eager type schemes. In *TAPSOFT '97: Theory and Practice of Software Development*, pages 490–501. Springer Verlag LNCS Vol. 1214, 1997.
23. C. Liang. A deterministic shift-reduce parser generator for a logic programming language. In *Computational Logic - CL 2000*, Springer-Verlag LNAI no. 1861, pages 1315–1329, July 2000.
24. C. Liang and G. Nadathur. Trade-offs in the intensional representation of lambda terms. Submitted for publication.
25. D. Miller. A logic programming language with λ-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.
26. D. Miller. Unification of simply typed lambda-terms as logic programming. In *8th International Logic Programming Conference*, pages 255–269. MIT Press, 1991.
27. D. Miller. Abstract syntax for variable binders: An overview. In *Computational Logic - CL 2000*, Springer-Verlag LNAI no. 1861, pages 239–253, July 2000.
28. D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

29. G. Nadathur. An explicit substitution notation in a λProlog implementation. Technical Report TR-98-01, Department of Computer Science, University of Chicago, January 1998. Also appears in the Proceedings of the First International Workshop on Explicit Substitutions, Tsukuba, Japan, March 1998.
30. G. Nadathur. A fine-grained notation for lambda terms and its use in intensional operations. *Journal of Functional and Logic Programming*, 1999(2), March 1999.
31. G. Nadathur and D. Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827. MIT Press, August 1988.
32. G. Nadathur and D. Miller. Higher-order logic programming. In D. Gabbay, C. Hogger, and A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 499–590. Oxford University Press, 1998.
33. G. Nadathur and D. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of λProlog. In *Automated Deduction–CADE-13*, Springer-Verlag LNAI no. 1632, pages 287–291, July 1999.
34. R. Paul. *Sparc architecture, assembly language programming, and C.* Prentice Hall, second edition, 2000.
35. F. Pereira and D. Warren. Definite clause grammars for language analysis. *Artificial Intelligence*, 13:231–278, 1980.
36. F. Pereira and D. Warren. Parsing as deduction. In *21st Annual Meeting of the Association for Computational Linguistics*, pages 137–144, 1983.
37. F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. D. Plotkin, editors, *Logical Frameworks.* Cambridge University Press, 1991.
38. F. Pfenning and C. Elliot. Higher-order abstract syntax. In *ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, 1988.
39. A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, 1993.
40. M. Wand. Correctness of procedure representations in higher-order assembly language. In *Proceedings: Mathematical Foundations of Programming Semantics '91*, pages 294–311. Springer-Verlag LNCS vol. 598, 1992.