An Overview of the Sloth2005 Curry System

System Description

Emilio Jesús Gallego Arias Julio Mariño
Universidad Politécnica de Madrid *
egallego@babel.ls.fi.upm.es jmarino@fi.upm.es

Abstract

We present the current state and immediate future developments of Sloth [7], a Curry [2] to Prolog translator. Currently it implements almost all the features required the *Curry Report* [5] – no encapsulated search at the moment – and there is support for some extensions like, for instance, *type classes* and *constraint programming* over rationals.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.2 [*Programming Languages*]: Language Classifications—Applicative Languages, Constraint and Logic Languages, Multiparadigm Languages

General Terms Languages

Keywords Curry, Sloth, Type Classes, Constraint Definitional Trees

1. Introduction

Sloth is a compiler that translates Curry programs into Prolog, extending our previous work on the translation of Babel programs.

The motivation for implementing and maintaining an apparently inefficient implementation of Curry is the need of keeping up to date with a rapidly evolving language, easily introducing changes that would take longer in an abstract machine implementation. Nevertheless, while slower than Prolog, Sloth is perfectly usable as a first contact with Curry.

Another goal is to encourage other groups to have their own implementations of Curry, by making the front-end of Sloth available to them.

Currently, Sloth generates Ciao Prolog [6] code but we think is 100% ISO compliant - i.e. easily portable. One of the main advantages of Sloth being built on top of Ciao is the huge number of libraries, extensions of LP, metaprogramming facilities, analyzers, etc, already present in the Ciao system which make life easier when trying some things. Moreover, Ciao is an open source project, so in case there is some feature lacking it is possible to add it — specially when the main development team is in the next room.

* The authors are totally/partly supported by the Spanish MCYT grant TIC2002-0055.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WCFLP'05 September 29, 2005, Tallinn, Estonia.
Copyright c 2005 ACM 1-59593-069-8/05/0009. . . $5.00.

Sloth2005 is available at `http://babel.ls.fi.upm.es/research/Sloth`.

The basic Curry to Prolog translation schemes can be found in our first paper [7], along with more on the Sloth concept. The following sections focus on specific features recently implemented or in development. Section 2 describes the translation of modules. The interactive shell is presented in Sec. 3. Section 4 describes the support of type classes, while Sec. 5 introduces constraint programming over the rationals. Finally, Sec. 6 presents some features currently being developed. In order to make the paper more self-contained, the appendix introduces the basics of the Curry to Prolog translation.

2. Modules support

We have based our module system on Ciao Prolog's. The underlying idea is to compile a Curry module to a Prolog module, and let Ciao's module system do almost all the work.

A compiled Curry module has two different parts:

Module interface This part tells the compiler what are the types and functions exported for use by other modules, and their respective types or constructors.

Object code This part mainly contains Prolog predicates to implement reduction to head normal form of Curry defined function symbols.

The key idea here is to have both parts in the same file, so we can handle a Curry module as a single file.

2.1 Identifier handling

All type and function identifiers are internally qualified, using the following format: `qid(ModuleName, IdentifierName)`.

When compiled to Prolog, we use the following translation scheme:
`qid(Module, Identifier)` → `Module_Identifier`.

2.2 Interface handling

The module's interface is represented by a Prolog predicate `exports/1`, for each symbol exported. The predicate can contain two different terms:

A qualifier exporter term:
`exports(id(IdName, Type))` where `IdName` is a qualified identifier and `Type` is the identifer type.

A type exporter term:
`exports(type(Type, [Constructor]))`, where
`Constructor = (CName, CType)`.

Interface loading is done by importing the `exports/1` predicate in the compiler, and then finding them by qualified name:

```
load_exports(Module) :-
        use_module(Module, [exports/1]),
        !,
        findall(iface(Iface), Module:exports(Iface), Exports
            ),
        load_interfaces(Exports).
```

The reader can see how powerful Ciao's module system is in the above code. Predicate `use_module/2` loads the `exports/1` predicate and bring it into the scope, then all the exports of the module are found and passed to the `load_interfaces/1` predicate, which loads all the exports into the compiler's symbol table.

The programmer is allowed to specify external functions (generally programmed in Prolog) using the `external` keyword. This will generate the interface for the function without any code:

```
(+) :: Int → Int → Int
(+) external
```

A similar scheme is used in the PACKS Curry compiler.

2.3 Code handling

Code sharing is allowed by making the `tohnf/2` predicate multifile, so the only thing a module importing `Module` has to do is `use_module(Module)` in its Prolog code, and the `Module`'s code will be accessible.

3. Interactive Shell

One of the most wanted features of any declarative system is an interactive shell. Given that our compiler translates Curry code to Prolog code, the natural way of implementing an interactive shell is generating a Prolog query from shell expressions and executing it against the loaded expressions.

3.1 Compiling expressions

Expression compilation is done by the non-interactive parser and translator, without using any especial code.

3.2 Executing code

Once we have the expression in Prolog (which is our internal representation), we turn it into a query and execute it:

```
exec(Expr, (Binds, Res)) :-
        free_rule(Expr, ExprFree, Dicc),
        tr_expr(ExprFree, Prolog),
        varset(Prolog, Vs1),
        filter_bindings(Vs1, Dicc, Binds),
        Query = tonf(Prolog, Res),
        (
            call(Query)
        ;
            Res = failed
        ),
        !.
```

Required modules for the execution are imported using the module system.

Some steps of our code translation scheme (as lambda lifting) result in new auto generated functions. This is solved in two ways:

Make `tohnf/2` a dynamic predicate: This would be the preferred solution, but debugging capabilities and performance of the resulting Prolog code are affected in serious ways[1]. We think Prolog compilers will be ready for such advanced use in little time, so this will be the final solution we will adopt.

Compile all the additional functions into a temporary module and load it: This is more a workaround than a real solution, but the performance and ease of implementation suppose a valuable edge about the dynamic solution.

[1] Talking with Ciao developers, the main blocker for turning `tonhf/2` into a dynamic predicate is that global analysis and optimization are disabled on said predicates, missing the predicate pre-indexing optimization, which is fundamental to our implementation due that Sloth relies in the efficient unification of the first parameter of `tohnf/2`.

3.3 Partial evaluation

One of the most attractive features of the shell is the possibility of performing a partial evaluation of the current expression.

Normally, a query E is translated to a Prolog query in the form of `tonf(E, Result)`. Partial evaluation is achieved by modifying the translation scheme such as every case of a `tonf/2` reduction the execution stops and the shell is awaken so it can show the results, translating back the Prolog expression to Curry.

This is an experimental feature at the moment, but we plan to formalize it and define the semantics for this kind of partial evaluation.

4. Type classes

Type classes are one of the distinguishing features of Haskell and, of course, they are expected to be seriously supported by the different implementations of Curry. We are currently supporting their most basic incarnation – no constructor classes, no multiparameter classes, etc.

4.1 Type checker design

The previous Sloth type checker was based on a classical Prolog implementation of a Hindley-Milner type inference algorithm. One of its key features is, of course, the representation or type variables by Prolog variables, and to rely on Prolog's unification algorithm for the type extraction process.

Adapting our type checker to allow type classes has been done using attributed variables. Actual class constraints are stored in the variable's extended attributes. This way, when unifying two types, the type checker can perform the type classes verification, without any modification on the core algorithm.

4.2 Code translation

We follow the translation scheme presented in [3], which proposes translating all the expressions involving a type class overloading function to an equivalent program whose types belong to a classical Hindley-Milner type system.

This implies adding another parameter to any function that is defined by a type class, but provides a significant advantage in terms of ease of implementation and correctness.

5. Constraint Programming

One natural extension to functional–logic programming is using constraints. We have done a very basic CLPQ implementation in Sloth, using the CLPQ library from CIAO.

5.1 Allowed operators

The newly introduced operators are:

`:+: :: Num a ⇒ a → a → a`
Addition over $\mathbb{Q}$.

`:-: :: Num a ⇒ a → a → a`
Subtraction over $\mathbb{Q}$.

`:*: :: Num a ⇒ a → a → a`
Multiplication over $\mathbb{Q}$.

`:/: :: Num a ⇒ a → a → a`
Division over $\mathbb{Q}$.

`:<: :: Num a ⇒ a → a → Success`
Less than over $\mathbb{Q}$.

`:<=: :: Num a ⇒ a → a → Success`
Less than or equal over $\mathbb{Q}$.

`:>: :: Num a ⇒ a → a → Success`
Greater than over $\mathbb{Q}$.

`:>=: :: Num a ⇒ a → a → Success`
Greater than or equal over $\mathbb{Q}$.

5.2 Implementation

All the CLPQ code is placed into the libraries, the compiler doesn't know anything about it.

The libraries are written in pure Prolog, using CIAO's `clpq` package.

This extension is highly experimental and will change in the future, perhaps to be compatible with PACKS [4]. A first step will be lifting our constraint support from the domain of $\mathbb{Q}$ to $\mathbb{R}$.

6. Planned improvements

We think Sloth is rapidly evolving into a mature system, and as such we are planning some future improvements.

Two main areas of future work are *Constraint Definitional Trees and* Encapsulated Search, when completed will make Sloth compliant with the Curry Prelude and establish a solid base for future research.

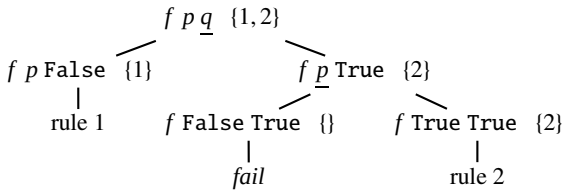
6.1 Constraint Definitional Trees

CDTs [8] are a proposal by the second author and José María Rey to extend the current compilation scheme of needed narrowing from sets of pattern-based definition rules to sets of constraint-guarded rules. The following lines assume certain familiarity with definitional trees – see [1] for a recent survey on the subject.

For instance, a definition like the following one will be allowed and a sequential, lazy code can be generated for it:

```
f :: Bool → Bool → Bool
f p q | ¬ q   = True
      | p ∧ q = False
```

Variable `q` is a pivot since substituting $\perp$ for it makes both guards undefined (that is, both guards demand `q`). A branch node at position {2} is thus generated. Since `q` ranges over {`False`, `True`}, two branches are generated (one for each value). When `q` is `False`, guard 1 is taken for further processing. Guard 2 is chosen when `q` is `True` (right branch of the tree). This branch continues in another branch node since `p` is demanded by the second guard. Again, two branches come out from this node. The resulting CDT can be depicted as:



where the positions for branching (pivots) have been underlined and the sets of (indices of) applicable cases are shown between braces.

6.2 Encapsulated Search

Support for encapsulated search is still an open problem. At the moment, a provisional solution using Ciao's threads library is on the way, but we are still looking for more clever translation schemes.

Acknowledgments

The authors wish to thank the anonymous referees for their useful comments.

References

[1] Sergio Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40(1):875–903, 2005.

[2] M. Hanus (ed.), H. Kuchen, and J.J. Moreno-Navarro. Curry: An integrated functional logic language. Technical report, RWTH Aachen, 2000.

[3] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996.

[4] Michael Hanus. *Packs User's Manual*. Available at `http://www.informatik.uni-kiel.de/~pakcs/`.

[5] Michael Hanus, Sergio Antoy, Herbert Kuchen, Francisco J. López-Fraguas, Wolfgang Lux, Juan José Moreno-Navarro, and Frank Steiner. *Curry: An Integrated Functional Logic Language*, 0.8 edition, April 2003. Editor: Michael Hanus.

[6] M. Hermenegildo, F. Bueno, M. García de la Banda, and G. Puebla. The CIAO multi-dialect compiler and system: An experimentation workbench for future (C)LP systems. In *Proceedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming*, Portland, Oregon, USA, december 1995. Available from `http://www.clip.dia.fi.upm.es/`.

[7] Julio Mariño and José María Rey. The implementation of Curry via its translation into Prolog. In Kuchen, editor, *7th Workshop on Functional and Logic Programming (WFLP98)*, number 63 in Working Papers. Westfälische Wilhelms-Universität Münster, 1998.

[8] Julio Mariño and José María Rey. Adding constraints to curry via flat guards. In *Workshop on Curry and Functional Logic Programming, WCFLP2005*. ACM, 2005. To appear.

A. Essentials of the Curry to Prolog translation scheme

In this section we present a very brief and informal description of the translation method used by Sloth. A complete reference can be found in [7].

A.1 Main execution model

The execution of Curry is achieved in Sloth by lazy narrowing. This reduction strategy is based on computing the Head Normal Form of an expression.

An expression E is in Head Normal Form when:

- E is a free variable.
- E is a constant.
- E is a constructor.
- E is a partial applied function.

Any other expression E, like a function application, is not in Head Normal Form, so our compiler generates for it a `tohnf/2` predicate, which takes as the first parameter the expression to evaluate, and unifies the second with the given in Head Normal Form.

Let's look to the `if_then_else` function:

```
if_then_else :: Bool → a → a → a
if_then_else True  t _ = t
if_then_else False _ f = f
```

We can see that this function needs the first parameter to be in Head Normal Form, and depending on the parameter value, it should return the second or third parameter.

The desired Prolog code is:

```
tohnf(if_then_else(B,C,D), Res) :-
        tohnf(B, E),
        if_then_else1(E,C,D,A).

if_then_else1('True',A,_,B) :-
        tohnf(A,B).
if_then_else1('False',_,A,B) :-
        tohnf(A,B).
```

The first step that the `tohnf/2` predicate does is to evaluate to Head Normal Form the first parameter, and then it calls the corresponding rule. Pattern matching is implemented using standard Prolog unification. More details on how the compiler knows what parameters are needed will follow in the next section.

Note that the second and third parameter are not guaranteed to be in Head Normal Form, so `if_then_else1` has to reduce those parameters to HNF before returning the result.

A.2 Dealing with lazy evaluation

Curry is a lazy language, so our compiler has to detect which values are needed, to avoid any unnecessary computation. The main place where this occurs is when a function tries to evaluate its parameters.

Sloth solves this problem using Definitional Trees. Definitional Trees are able to obtain the needed parameters and the evaluation order for them, so we can guarantee that no unnecessary parameter evaluation will take place.

The compiler implements Definitional Trees using a two phase approach, first it reduces to Head Normal Form the first needed parameter and then it calls the pattern matching rule:

```
tohnf('Prelude_length'(B), A) :-
        tohnf(B,C),
        'Prelude_length1'(C,A).

'Prelude_length1'([],0).

'Prelude_length1'([_|A],B) :-
        tohnf('Prelude_+'(1,'Prelude_length'(A)),B).
```

This process is applied recursively when more than one parameter is needed by the function.

A.3 Evaluating to Normal Form

The evaluation to Head Normal Form not always gives the desired results, because it doesn't force any of the constructor parameters to be in Head Normal Form. This yields partially evaluated terms, so we have to specify a stronger state for an expression, the Normal Form.

An expression E is in Normal Form when:

E is a free variable.

E is a constant.

E is a constructor and all its parameters are in Normal Form.

The Prolog predicate that implements this is called `tonf/2`.