A Haskell Nominal Toolkit

Christophe Calvès[1,2]

Department of Computer Science
King's College London
London, U.K.

Keywords: Binders, Haskell, Nominal, Rewriting, Zipper

1 Introduction

The notion of a binder plays a central role in computer science: logic formulas, programming languages are examples of systems that involve binders. Reasoning on theses systems requires to be able to reason up to α-equivalence classes. For example, in lambda calculus the terms $\lambda x.x$ and $\lambda y.y$ have the same semantics. We need to be able to distinguish between free and bound variables and algorithms have to work up to α-equivalence classes.

Nominal techniques were introduced to represent in a simple and natural way systems that include binders [10,14,15]. We can see nominal terms as trees with three kinds of leaves:

- *constants* f, g, h, ...: as first-order term constants.
- *variables* X, Y, Z, ...: meta-variables, they represent unknown terms.
- *atoms* a, b, c, ...: they are just *names*.

and three kinds of internal nodes: a *pair* (t_1, t_2) or a *modal* node which can be either: an *abstraction* $[a]t$ (it means that a is abstracted, it can be almost any atom with some conditions) or a *swapping* $(a\ b)\cdot t$ (it represents the term obtained by renaming every a by b and every b by a in t).

Let's consider the nominal term $(a\ b)\cdot[a](f,(a,X))$. One might think it is equivalent to $[b](f,(b,X))$, obtained by performing the swapping in $[a](f,(a,X))$ but it is not the case. Indeed X represents an unknown term, the swapping has also to be performed on it. Thus swappings are suspended on variables until they are substituted. $(a\ b)\cdot[a](f,(a,X))$ is actually equivalent to $[b](f,(b,(a\ b)\cdot X))$.

Let's take another example. The atom a is abstracted in $[a](f,(a,X))$. Because a can be allmost any atom, we would like it to equivalent to $[b](f,(b,(a\ b)\cdot X))$ obtained by swapping a and b in the term. Unfortunately, if we substitute X by b the two terms are not equivalent: the free name b in $[a](f,(a,b))$ became the free name a in $[b](f,(b,a))$. Nominal theory uses the notion of *fresh atoms*. An atom b is fresh in a term t (written $b \# t$)

[1] This work has been partially funded by the EPSRC grant "CANS" (EP/D501016/1). We would thanks Maribel Fernández, Andrzej Filinski and Oleg kiselyov

[2] Email: Christophe.Calves@kcl.ac.uk

if b does not occur unabstracted in t. In our example, $[a](f,(a,X))$ and $[b](f,(b,(a\ b)\cdot X))$ are equivalent if and only if b is fresh for X.

These considerations make nominal algorithms harder to develop than first-order ones. It requires to compute and perform swappings, take care that atoms are fresh for a term, compute the required freshness conditions, etc ... That is one of the reasons why we developed a *Haskell Nominal Toolkit(HNT)* [4] providing with it a whole monadic framework to handle automatically as many of nominal details as possible such as checking freshness conditions, performing swappings transparently, etc .... *HNT* includes also a set of "building blocks" such as α-equivalence and matching algorithms and abstraction functions to make nominal programming quite as easy as first-order. Finlay it is a rewriting framework providing nominal rewriting functions and a zipper to navigate with any term.

HNT comes from the idea that instead of developing a stand-alone tool or language providing some features for nominal terms, it would be easier and better to develop a nominal framework for an existing language. Because nominal techniques involve a lot of technical details, we wanted to use monads to handle as much details as possible transparently. Haskell appeared to be the best choice.

To illustrate the possibilities of *HNT*, we developed an interactive nominal rewriting tool. It is inspired by the *Zipper-based file server/OS* [13] of Oleg Kiselyov. There are many rewriting tools, such as *MAUDE* [7,2], *ELAN* [3]. These tools are very useful to define a rewriting system and strategies. But it is often useful when developing a rewriting system to try some rules on a term to see how the rules actually work one step at a time. The user has to be able to navigate within a term (as it would be a UNIX file system for example), replace any subterm and apply a rewrite rule on any subterm interactively. This is the purpose of *HNT*. To our knowledge, there is no other tool with the same interactive functionalities.

In section 2 we will show how the interactive tool works. In section 3 we will describe the library.

2 The Interactive Nominal Rewriting Tool

The best way to describe the interactive tool is to show a session. In the following, `a:1`, `a:2` and `a:3` will represent respectively the atoms a, b and c, `v:1` and `v:2`, the variables X and Y and `c:1` the constant f. We write the nominal term $[c][a](X,(a\ c)\cdot c)$ with no context `|- [a:3][a:1](v:1 , (a:1 a:3) a:3)` and input it directly when *HNT* is running:

```
Enter Term in Context : |- [a:3][a:1](v:1 , (a:1 a:3) a:3)
  Path = /
  Term = [a:3][a:1](v:1,(a:1 a:3)a:3)
  Ctxt =
=>
```

`Term` shows the current subterm, `Path` the path to this subterm and `Ctxt` the freshness context. We start at the root node of the term with an empty freshness context. We can move from a subterm to the next by the command `move next` (in a pair, it moves from left to right):

```
=> move next
  Path = //modal
  Term = [a:1](v:1,(a:1 a:3)a:3)
  Ctxt =
=> move next
...
=> move next
```

```
...
=> move next
  Path = //modal/modal/right
  Term = (a:1 a:3)a:3
  Ctxt =
=>
```

or go directly to the subterm we want:

```
=> move down
  Path = //modal
  Term = [a:1](v:1,(a:1 a:3)a:3)
  Ctxt =
=> move down
...
```

```
=> move down to right
  Path = //modal/modal/right
  Term = (a:1 a:3)a:3
  Ctxt =
=>
```

We can change the current subterm by any term using the command `update` :

```
=> update c:1
  Path = //modal/modal/right
  Term = c:1
  Ctxt =
=>
```

The rest of the term is unchanged:

```
=> move up
...
=> move up
  Path = //modal
  Term = [a:1](v:1,c:1)
  Ctxt =
=>
```

Another way to change a subterm is applying a nominal rewrite rule on it by the command `rewrite`. If the rule can be applied, the subterm is replaced by the result of the rule. Otherwise a message explaining the error is printed and nothing is changed. For example:

```
=> rewrite |- [a:2](v:1,v:2) -> v:2
Rule didn't match : Constraints not satisfied
  Path = //modal
  Term = [a:1](v:1,c:1)
  Ctxt =
=>
```

Indeed $[a](X, f)$ matches $[b](X, f)$ only if a and b are fresh for X. Because the context does not have these hypotheses, the left part does not match the subterm so the rule can not be applied. Let's change the context by the command `context` and try again:

```
=> context {a:1 a:2}#v:1
  Path = //modal
  Term = [a:1](v:1,c:1)
  Ctxt = {a:1 a:2}#v:1
=> rewrite |- [a:2](v:1,v:2) -> v:2
  Path = //modal
  Term = c:1
  Ctxt = {a:1 a:2}#v:1
=>
```

3 The Toolkit

HNT is above all a Haskell library providing a framework for nominal terms. It is composed of three main parts: a monadic framework handling nominal properties, nominal algorithms such as α-equivalence and matching and a rewriting framework.

3.1 The Monadic Framework

HNT makes massive use of monads. To make the implementation more flexible each effect is isolated into a single monadic layer. There are four different layers: the environment, the freshness context, the substitution and zipper layers.

The environment layer implements a current environment as in [5] with functions to compose a swapping to the current permutation, to add/remove atoms from the set of freshness constraints, perform the modify/restore operations, ... For example the code:

```
localSwapR a b (do b <- image a
                   u <- localSetFresh False a (f t)
                   return (Modal (Abs b) u)
               )
```

locally swaps a in b in current environment, gets the image of a by the current permutation, computes `f t` with a added locally to the current freshness set and finally returns `(Modal (Abs b) u)`. All the complicated details of swapping, modifying/restoring the environment are done transparently.

The freshness context and substitution layer implement respectively a current freshness context and a current substitution and simple functions to check if a constraint is in the context, add one to the context, assign a term to a variable in the substitution, etc ...

We often need several of these effects. For example solving a matching problem requires an environment, a freshness context and a substitution. Layers are implemented by Andrzej Filinski's *Monadic Layers* [9]. This approach embeds every monad in a continuation monad with state passing. The layers are stored in the state. The main benefit of this approach is all the code lives within only one monad and the layers in the state. It leads to simpler and neater code.

3.2 The algorithms

HNT implements five nominal algorithms. All these algorithms raise an error if the problem they try to solve does not have any solution. The algorithms are:

- an α-equivalence solver: `alpha'solve t u` returns the freshness context that makes t and u α-equivalent.
- an α-equivalence checker: `alpha'check fc t u` checks if t and u are α-equivalence with the freshness context fc.
- a matching solver : `match'solve fc l t` returns the freshness context Δ and the most general unifier σ such that $l\sigma \approx_\alpha t$ and every constraint in the freshness context fc is satisfied.
- a matching checker : `match'check (fc,l) (fc',t)` returns the most general unifier σ such that $l\sigma \approx_\alpha t$ and fc is satisfied under the freshness context fc'
- a rewriting function : `rewrite rl t` returns the term obtained by applying the rule rl on the term t.

The α-equivalence and matching solvers are based on the algorithms presented in [5]. These algorithms are basic blocks to build complex programs involving nominal terms as easily as if they were first-order terms. For example, to apply a rule `rl` as many times as possible on a term `t` :

```
rewriteMany rl t = catchError' (rewrite rl t >>= rewriteMany rl)
                               (\error -> output error >> return t)
```

where `catchError'` is the monadic function to catch an error in the corresponding monad.

3.3 The Zipper

The function `rewrite` performs only head-rewriting. To implement strategies we need to be able to rewrite anywhere in a term. A zipper is a structure invented by Huet [11] to navigate within a term, access and update efficiently a subterm of the term. It can be seen as a pointer to a subterm but it is pure. *HNT*'s zipper implementation is inspired by Oleg Kiselyov's *Generic Zipper and its applications* [12], but instead of the powerful but complex *Monadic Framework for Delimited Continuation* [8] and a term traversal function that can be difficult to write, we use the usual continuation monad and a simple function that gives, for a term, the list of its subterms and their context. Everything that is possible with the interactive tool is possible in exactly the same way within the toolkit. In fact the interactive tool is just a wrapper for the toolkit.

4 Related and Future Work

Related Work

There exists some papers and tools to deal with names: James Cheney implemented a Haskell library based on nominal theory named *FreshLib* [6], the *FreshML Reseach Project* [1] lead to *Fresh Objective Caml*, "a patch of the Objective Caml language that includes facilities for programming with names and binders", ... Though these projects address the "name" problem, their objectives and ways of doing are very different from *HNT*. In a nut shell, the goal of *HNT* is not to scrap your name plate but scrap your name-swappings-and-freshness-constraints-while-moving-in-your-nominal-term plate. Which means that *HNT*, on the contrary to *FreshLib* and *Fresh Objective Caml*, neither handles user-defined data types nor generates fresh names. But it gives complete control over name management, freshness checking, substitutions, ... For example, names are not an abstract data type as in *Fresh Objective Caml*, they are explicit types that can be manipulated at will.

Future Work

To simplify the development, we decided to use functional maps instead of mutable arrays in *HNT*, which introduces a logarithmic factor. However, *HNT* has been designed to support mutable arrays and switching to functional maps to diff arrays is straightforward and will have the same complexity as with mutable arrays (because we never backtrack). An efficient nominal unification algorithm is also being developed. We plan to include it in *HNT* as soon as possible.

5 Conclusion

The toolkit enables to program with nominal terms as easily as with first-order ones. Being a Haskell library, it benefits from all the power, the freedom and the existing libraries of Haskell. The rewriting framework, though being quite minimal, is very flexible and enables to program in a few lines complex strategies. The interactive tool automates part of the work that is usually done by hand. It enables to see interactively how a strategy operates on a term, which is helpful when designing strategies.

References

[1] Freshml research project website available at. http://www.cl.cam.ac.uk/~amp12/freshml/.

[2] The maude system website:. http://maude.cs.uiuc.edu/.

[3] P. Borovansky, C. Kirchner, H. Kirchner, P.E. Moreau, and C. Ringeissen. An overview of ELAN. In *Proceedings of the International Workshop on Rewriting Logic and its Applications*, volume 15, 1998.

[4] Christophe Calvès. Haskell nominal toolkit available at. http://www.dcs.kcl.ac.uk/pg/calves/hnt.

[5] Christophe Calvès and Maribel Fernández. Nominal matching and alpha-equivalence. In Wilfrid Hodges and Ruy J. G. B. de Queiroz, editors, *WoLLIC*, volume 5110 of *Lecture Notes in Computer Science*, pages 111–122. Springer, 2008.

[6] J. Cheney. Scrap your nameplate:(functional pearl). In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, volume 40, pages 180–191. ACM New York, NY, USA, 2005.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martı-Oliet, J. Meseguer, and JF Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[8] R.K. Dybvig, S.P. Jones, and A. Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 2005.

[9] Andrzej Filinski. Monadic reflection in haskell available at. http://cs.ioc.ee/mpc-amast06/msfp/filinski-slides.pdf.

[10] M.J. Gabbay and A.M. Pitts. A New Approach to Abstract Syntax with Variable Binding. *Formal Aspects of Computing*, 13(3):341–363, 2002.

[11] G. HUET. The Zipper. *Journal of Functional Programming*, 7(05):549–554, 1997.

[12] Oleg Kiselyov. Generic zipper and its applications available at. http://okmij.org/ftp/Computation/Continuations.html#zipper.

[13] Oleg Kiselyov. Zipper-based file server/os available at. http://okmij.org/ftp/Computation/Continuations.html#zipper-fs.

[14] A.M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186(2):165–193, 2003.

[15] M.R. Shinwell, A.M. Pitts, and M.J. Gabbay. FreshML: programming with binders made simple. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 263–274. ACM New York, NY, USA, 2003.