NEEDLE & KNOT: A Framework for Meta-Theoretical Specifications with Binding

Steven Keuchel
Ghent University
steven.keuchel@ugent.be

Klara Mardirosian
KU Leuven
klara.mar@cs.kuleuven.be

Tom Schrijvers
KU Leuven
tom.schrijvers@kuleuven.be

1. Introduction

This extended abstract showcases NEEDLE & KNOT, a framework for programming language mechanization. Our framework tackles the problem that formal proofs about programming language semantics and type-systems are long and error-prone. To guarantee their correctness, techniques for mechanical formalization in proof assistants, such as ones based on type-theories, have received much attention in recent years. However, human mechanizers of programming language meta-theory are unnecessarily burdened with boilerplate that arises from variable bindings in programming languages. To tackle the substantial boilerplate for richer languages, support from the proof assistant or from tools is mandatory.

NEEDLE & KNOT specifically target the boilerplate that arises in type-safety proofs of typed programming languages. In particular, they use the syntactic approach to programming language metatheory, invented by Wright and Felleisen [13] and popularized by Pierce [10]. The treatment of variable binding typically comprises the better part of such formalizations. Most of this variable binding infrastructure is repetitive and tedious boilerplate. By boilerplate we mean mechanical operations and lemmas that appear in many languages, such as: 1) common operations like calculating the sets of free variables or the domain of a typing context, appending contexts and substitutions; 2) lemmas about operations like commutation of substitutions or the interaction between the free-variable calculation and substitution; and 3) lemmas about semantic relations such as the preservation of well-scoping and typing under operations.

Our approach consists of two ingredients: 1) KNOT, a domain-specific specification language for abstract syntax and relational semantics of programming languages and 2) NEEDLE, a tool to generate COQ code from KNOT specifications. Our case studies demonstrate that this approach significantly reduces the size of language mechanizations and enables the mechanizer to skip the tedious boilerplate and directly work on the interesting meta-theory.

2. KNOT Specifications

This section illustrates KNOT on a specification of the simply-typed lambda calculus (STLC). More elaborate specifications can be found on our project page [1] and in the papers [7, 8].

2.1 Syntax Specifications

The following code snippet shows the KNOT declarations for the syntax of STLC:

```
namespace Var : Tm
sort Ty :=
 | tarr (T1 T2: Ty)
 | tunit
```

```
sort Tm :=
 + var (x@Var)
 | abs (x:Var) (T: Ty) ([x]t: Tm)
 | app (t1 t2: Tm)
 | tt
```

We start with the declaration of a namespace `Var` for term variables, which is followed by the declarations of the two sorts of STLC: simple types and terms. [1]

Variable bindings are declared via binding specifications that can be inserted in square brackets `[]` before a field. In the example the abstraction case `abs` is the only one that contains a binding specification `[x]t` which denotes that the variable `x` is brought into scope in the body `t`. KNOT also supports richer binding specifications, e.g. it allows to bind an arbitrary amount of variables for instance defined by binding forms such as declaration lists or typed patterns [8].

The variable constructor `var` is specifically denoted as such by introducing it with a `+` rather than than a `|`. Moreover, the variable `x` appears in a different mode in the `var` constructor than in the `abs` constructor: the λ-abstraction binds the variable `x` and we call it a *binding occurrence* whereas the `x` in the variable constructor is a *reference* or *use occurrence*.

Requiring the variable constructor to have no other fields than a single reference and disallowing all other constructors to have references essentially enforces a free monad-like structure on the syntax which allows NEEDLE to derive the substitution function fully generically.

2.2 Semantic Specifications

In addition to the specification of abstract syntax, KNOT also supports the definition of semantic relations like typing or reduction. The following code extends the example specification in Section 2.1 with the typing relation for STLC:

```
env Env :=
  + empty
  | evar : Var -> Ty : Typing
relation [Env] Typing Tm Ty :=
  + Tvar : {x -> T} -> Typing (var x) T
  | Tabs : [x -> T1] Typing t (weaken T2 x) ->
           Typing (abs x T1 t) (tarr T1 T2)
  | Tapp : Typing t1 (tarr T11 T12) ->
           Typing t2 T11 -> Typing (app t1 t2) T12
  | Ttt :  Typing (tt) (tunit)
```

The first declaration introduces typing environments `Env` that map term variables to their associated types. It also states that the term variable binding in the environment are substitutable for `Typing` judgements. The typing relation `Typing` contains a rule for each constructor of the `Tm` sort.

The rule `Tvar` takes care of the variable case, and like variable constructors it is specifically designated as such using a `+`. The typing environment `Env` is only mentioned in the head of the decla-

[1] Namespaces and sorts are a distinct concept in KNOT. It is possible to declare multiple namespaces for the same sort, but usually there is at most one namespace per sort.

ration but is subsequently left implicit. Here the braces `{}` denote a lookup in the implicit environment. The rule simply fetches the typing information for the variable from the environment.

The rule `Tabs` for term abstractions `(abs x T1 t)` needs to add a binding for the `x` variable to the implicit environment for the typing of the body `t`. Only the difference is recorded: the binding `[x -> T1]` is added to the environment. Furthermore, the domain type `T2` changes scope and needs to be explicitly weakened in the premise.

In order for NEEDLE to derive the substitution lemma for this relation we need to employ similar restrictions as in Section 2.1. The variable constructor is required to only have a single lookup as a premise that match the indices of the conclusions and no other rules are allowed to contain lookups, or stated more generally, to inspect the outer environment in any way.

3. The NEEDLE Tool

KNOT comes with a code generator tool, NEEDLE, that compiles KNOT specifications to specicalized COQ code. It uses a de Bruijn representation that has been sufficiently generalized to deal with multiple namespaces, e.g. to support truly heterogeneous binders [8]. NEEDLE uses various syntax-directed elaborations to generate functions for syntactic operations, theorems statements and proof terms. Only lemmas that do not depend on the user-specified abstract syntax are handled by (generic) proof tactics.

The kind of generated boilerplate includes:

1. Shifting, substitution and size functions.
2. Commutation and cancellation lemmas.
3. Well-scopedness predicates.
4. Shifting and substitution lemmas for well-scopedness.
5. Environment lookups.
6. Well-scopedness lemmas for lookups.
7. Well-scopedness lemmas for relations.
8. Shifting and substitution lemmas for relations.

NEEDLE also comes with a tactic library for automatically discharging proof obligations that frequently come up in proofs of weakening and substitution lemmas of type-systems. The code generator produces the necessary hints.

Not all type systems meet the restrictions of Section 2.2, e.g. algorithmic type systems usually have specialized variable rules, and the substitution lemma may not be fully derivable. In these cases NEEDLE leaves minimal proof obligations for the human mechanizer to fill in the gaps.

4. Case Studies

We have performed a case study of type-safety mechanizations, that we regularly update, to demonstrate the benefits of the KNOT approach. It allows us to make two complimentary evaluations. The first considers different mechanizations for the same language (the POPLMARK challenge) authored by different people with different degrees of automation or tool support. The second compares KNOT against manual mechanizations (written by the same author in a consistent and highly automated style) across different languages.

POPLmark Comparison Because it is the most widely implemented benchmark for mechanizing metatheory, we use parts 1A + 2A of the POPLMARK challenge to compare our work with that of others [4, 5, 9, 11, 12]. These parts prove type-safety for System $F_{<:}$ with algorithmic subtyping. Among these, the KNOT-based solution is with 192 SLOC by far the smallest, comprising half the SLOC of the next smallest solution that we are aware of, which is based on the AUTOSUBST [11] library.

Manual vs.* NEEDLE *Mechanizations The previous comparison only considers the type-safety proof for a single language, and thus paints a rather one-sided picture of the savings to be had. For this reason, our second comparison considers the savings across 11 languages. For each language, we have two COQ formalizations: one developed without tool support and one that uses NEEDLE's generated code. As their mechanizations are not readily available across different tools and systems, we here pit KNOT & NEEDLE only against our own manual mechanizations. To yield representative results, all our manual mechanizations have been written by the same author in a consistent and highly automated style.

The 11 textbook calculi we consider range from the simply typed lambda calculus to variants of System F that include mixes products with nested pattern matching, existentials, sub-typing and type operators.

5. Ongoing and future work

In an effort to precisely characterize the class of programming languages that KNOT supports and to further extend it, our current work focuses on a thorough understanding of the algebraic foundations of NEEDLE and KNOT. This will also increase the confidence in the correctness of our approach.

Previous work on the foundations of syntax with binders includes Fiore et al.'s [6] approach based on initial algebra semantics. Altenkirch et al. [2, 3] use *relative monads*, a generalization of monads, to model intrinsic well-scoping and well-typing relations in such a way that the substitution lemma coincides with the relative monad's bind.

Our goal is to adapt Altenkirch et al.'s results to more traditional extrinsic relations. Furthermore, we aim at extending these results to a generic universe of languages and relations.

References

[1] Needle & Knot Project Page. `https://users.ugent.be/~skeuchel/knot`. Accessed: 2016-11-28.

[2] T. Altenkirch, J. Chapman, and T. Uustalu. Monads need not be endofunctors. In L. Ong, editor, *Foundations of Software Science and Computational Structures*, volume 6014 of *LNCS*. Springer, 2010.

[3] T. Altenkirch, J. Chapman, and T. Uustalu. Relative monads formalised. *JFR*, 7(1), 2014.

[4] B. Aydemir and S. Weirich. LNgen: Tool support for locally nameless representations. Technical report, UPenn, 2010.

[5] A. Charguéraud. `http://www.chargueraud.org/softs/ln/`. Accessed: 2015-07-02.

[6] M. P. Fiore, G. Plotkin, and D. Turi. Abstract Syntax and Variable Binding, 2003. Extended Abstract.

[7] S. Keuchel, T. Schrijvers, and S. Weirich. Needle & Knot: Binder Boilerplate Bound Tighter. Unpublished Draft.

[8] S. Keuchel, S. Weirich, and T. Schrijvers. Needle & Knot: Binder Boilerplate Tied Up. In *ESOP*. Springer, 2016.

[9] G. Lee, B. C. Oliveira, S. Cho, and K. Yi. GMeta: A generic formal metatheory framework for first-order representations. In *ESOP*. Springer, 2012.

[10] B. C. Pierce. *Types and Programming Languages*. MIT press, 2002.

[11] S. Schäfer, T. Tebbi, and G. Smolka. Autosubst: Reasoning with de bruijn terms and parallel substitutions. In X. Zhang and C. Urban, editors, *ITP'15*, LNAI. Springer, 2015.

[12] J. Vouillon. A solution to the poplmark challenge based on de bruijn indices. *JAR*, 49(3), 2012.

[13] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1), 1994.