Extracting Theory Graphs from Aldor Libraries

Florian Rabe[1] and Stephen M. Watt[2]

[1] Computer Science, University of Erlangen-Nuremberg, Germany
[2] David R. Cheriton School of Computer Science, University of Waterloo, Canada

florian.rabe@fau.de and smwatt@uwaterloo.ca

Abstract. Aldor is a programming language for computer algebra that allows natural expression of algebraic objects while also allowing compilation to efficient code. Its language primitives, however, do not correspond exactly to those of modern proof assistants nor to those of data formats used in mathematical knowledge management. We discuss these difficulties and export the Aldor library as a diagram in the category of theories and theory morphisms, using a simplified model of the Aldor language that retains its essential expressivity. This allows us to capture a rich set of expert-designed interfaces for use in mathematical knowledge management settings.

Background

Aldor emerged from the Scratchpad II project at IBM Research, developed as a generalization of a language first described in [3] and known first as $A^\sharp$ [9], and the *Axiom Library Compiler* before its release as Aldor as an independent package. Types are run-time values, with run time *domains* providing abstract data types, and *categories* qualifying domains by requiring certain operations or properties. The application to symbolic mathematical computation influenced the design to use dependent types pervasively, conditional run-time category membership, and *ex post facto* type extension [7,8].

Theory graphs are categorical diagrams of theories using truth-preserving compositional interpretations as morphisms between theories. They are an important language-independent tool for high-level knowledge representation, interrelating diverse constructs, and modular theory development [2]. A critical choice in the design of formal languages for mathematics is whether theories and morphisms are provided by a meta-layer formalism (which is always possible and relatively straightforward) or built into the language as first-class objects (which greatly increases both expressivity and complexity). For mathematical knowledge, the *built-in* design is very appealing because it enables using theories as the types of mathematical structures, thus elegantly capturing mathematical practice. Thus, many systems choose it, including Aldor. But the *meta-layer* design is superior for integrating developments across different languages, systems, and libraries because they can share the theory layer, which provides exactly the interfaces needed for interoperability. This poses a recurring challenge for the integration of mathematical software systems.

In this paper, we present (i) a system for translating Aldor libraries into the language-independent theory graph formalism provided by MMT [6], and (ii) the data obtained by translating the available Aldor libraries in this way. The result comprises 440 theories and morphisms. This is valuable because (i) it makes the (so far unpublished) Aldor libraries accessible to the general community and (ii) provides insights into the general issue of connecting the *built-in* and *meta-layer* design choices. In particular, our work can serve as a starting point for porting the Aldor library to or as an interface for integrating Aldor computations in other mathematical software systems.

Modeling Aldor in MMT

As a running example we use the definition in Figure 1, based on $\sum^{it}$ [1]. Here the function `ResidueClassRing` takes two arguments `R` and `p` and returns a theory (called a *category* in Aldor). Theories are used as types akin to record types, and their elements (called *domains* in Aldor) provide definitions for all abstract fields of the category. Each Aldor domain provides a representation type (written `%`), which corresponds to a carrier set. The name of a domain doubles as a reference to that representation type, mimicking the mathematical practice of using the same name for a structure and its carrier. Here `R` is a domain, typed by a previously defined category, and `p` is an element of the carrier of `R`.

```
define ResidueClassRing(
    R: CommutativeRing, p: R): Category  ==
CommutativeRing with {
    modularRep: R -> %;
    canonicalPreImage: % -> R;
    if R has EuclideanDomain then {
        symmetricPreImage: % -> R;
        if R has SourceOfPrimes then
            if prime? p then Field;
    } }
```

Fig. 1. ResidueClassRing in Aldor

The category is defined to extend the category `CommutativeRing` with several declarations. Critically, Aldor allows conditional declarations: if `R` additionally has category `EuclideanDomain` (an extension of `CommutativeRing`), then *ResidueClassRing* is defined to also declare `symmetricPreImage`, and if `R` additionally has category `SourceOfPrimes`, *ResidueClassRing* is defined to also include the category `Field`. For example, `ResidueClassRing(Integer, 7)` extends `Field` with all three listed operations because the domain `Integer` has those two categories.

Representing Aldor Primitives We use a manually written MMT theory `Aldor` to declare the about 30 primitive operators of Aldor.[3] The theory `Aldor` occurs as the governing language (called *meta-theory* in MMT) of all theories we generate from Aldor libraries.

For simplicity, we have not formalized Aldor in a logical framework, instead we declare it directly as a primitive language in MMT. Therefore, the constants

[3] This theory can be found at https://gl.mathhub.info/aldor/language/.

in `Aldor` are untyped and provide only notations (via the `#` symbol). For example, we declare an operator `Qualify # L1 $ 2` for Aldor's primitive operation `modularRep $ C` of accessing the field `modularRep` of some domain variable `C` of category `ResidueClassRing`. MMT's notation language is expressive enough to mimic many details of Aldor's concrete syntax such as the `$`-notation for qualified names. These MMT constants occur as the heads of the MMT expressions representing Aldor expressions. For example, we export the expression above as the following MMT object (here given in OpenMath XML syntax):

```
<OMA><OMS cd="aldor" name="Qualify"/>
     <OMS cd="ResidueClassRing" name="modularRep"/><OMV name="C"/></OMA>.
```

Interpreting Categories and Domains as Theories and Morphisms Category-valued Aldor functions become parametric theories in MMT. For example, Fig. 2 shows the HTML+MathML rendering produced by MMT from our export of our example theory. The two kinds of declarations in Aldor categories (typed constants and category extensions) can be directly represented using the analogous features of MMT.

If a domain is declared at top level, it can be seen as a theory morphism from its type (which must be a category and thus be represented as an MMT theory) to the empty theory, i.e., the MMT theory `Aldor`. More generally, a domain-returning function can be represented as a theory morphism into the anonymous MMT theory declaring the function's arguments. If the type of a domain is the union of some Aldor categories, the domain of the MMT morphism is the corresponding union of MMT theories. If the type is an anonymous category, we generate a name for it and add it to the MMT theory graph. Finally, Aldor allows domains typed by an anonymous category and no definiens; this is Aldor's way of grouping statically available constants. We represent such domains as MMT theories.

ResidueClassRing
include CommutativeRing
constant modularRep
type $R \longrightarrow \%$
constant canonicalPreImage
type $\% \longrightarrow R$
constant symmetricPreImage
type $\vdash (R \text{ Has EuclideanDomain}) \longrightarrow (\% \longrightarrow R)$
theory EuclideanDomain
constant condition
type $\vdash (R \text{ Has EuclideanDomain}) \wedge (\vdash (R \text{ Has SourceOfPrimes}) \wedge \vdash (\text{prime?}\ p))$
include Field

Fig. 2. ResidueClassRing in MMT

The Implicit Carrier Type The type `%` is built into every Aldor category and treated specially by the system. We represent this in MMT by manually writing a special theory `Carrier` that declares only a type `%`. This theory is then included by every theory generated from an Aldor category. In Fig. 2, this include is not present explicitly because it is already inherited from `CommutativeRing` by composition of inclusions.

Each domain must define the name `Rep` to define the carrier set. In the language of theory graphs, this is simply the assignment to the constant `%`, and we translate it accordingly in the MMT theory morphism. To represent the Aldor type system correctly, we have to add a coercion rule that turns every use of an Aldor domain `R` in a position where a type is expected into the expression `%$R`.

Categories as Types Contrary to Aldor categories, Mmt theories cannot be meaningfully used as types directly. In fact, Mmt does not impose any type system at all. Instead, it is the task of the meta-theory to declare appropriate constants and typing rules. We have previously presented a solution for using theories as types in [5], and we follow the same approach here: the constant *Category* of the theory `Aldor` serves as the type of categories, and we provide the rules

$$\frac{\text{theory } T \text{ includes } \texttt{Carrier}}{T : \texttt{Category}} \qquad \frac{c : \texttt{Category}}{c : \texttt{type}}$$

to turn each category (including those that are created dynamically) into a type. Thus, we can represent the type of the variable `R` in the running example simply as an OpenMath object referencing the theory `ResidueClassRing`.

Conditional Declarations Mmt does not allow for conditional declarations, and intentionally so because that makes it statically undecidable what the names provided by a theory are. We found two novel solutions to encode Aldor's conditional declarations that will have to be evaluated in the future.

First, we extend our representation of Aldor's type system with a propositions-as-types principle and represent the condition as an additional argument. For example, in Fig. 2 the condition of the constant `symmetricPreImage` is represented by the type $\vdash$ `R Has EuclidenDomain` (which uses Aldor's built-in operator `Has` for testing if a domain implements a category). This requires providing a proof every time the constant is used. In Aldor these proof obligations are discharged by direct computation, which amounts to a very simple sound-but-incomplete theorem prover. Therefore, we do not synthesize proof terms for them and instead generate a placeholder for an unknown subterm to be reconstructed by the reusing application.

This approach does not work for conditional includes though. Therefore, we developed a second solution that is more involved but would also be applicable to for every conditional declaration, we produce a nested theory that declares first the condition as an axiom and then the actual declaration. For example, in Fig. 2 the inclusion of `Field` is guarded by the conjunction of the two conditions in whose scope it occurs. We need to generate a name for this new theory, and we use some heuristics to pick a helpful name for it, in this case `EuclideanDomain`. Now given a proof of the condition, we can construct a theory morphism from the nested theory into its parent, via which the conditional declarations can be accessed.

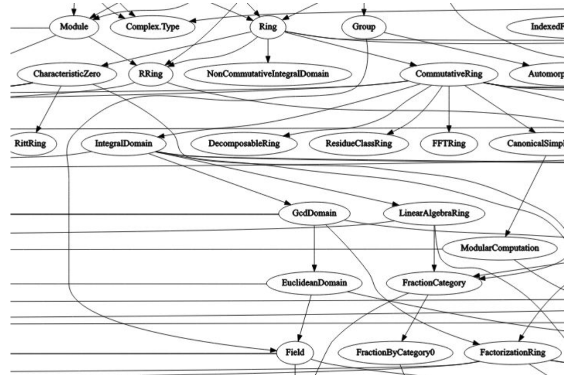

Fig. 3. ResidueClassRing theory graph fragment

Representing Aldor Libraries as MMT Theory Graphs The Aldor distribution includes six libraries, but we focus on the two that have the most reuse value: the base library and the algebra library. We follow the best practice [4] of exporting a system-near export that is then imported into the target system using a general purpose data format (in our case: Lisp S-expressions) as an intermediate representation. We have adapted the Aldor compiler to produce the intermediate representation (as one `.axy` file for each of the 321 Aldor source files available), and we have written an MMT import tool that implements the representation described above. The (very verbose) intermediate representation makes up 360 MB (18 MB gzipped). This yields 440 MMT theories and morphisms, written out as OMDoc files totaling 0.5 MB, which is similar in size to the zipped Aldor sources. The import time is on the order of 1 minute on modern laptops. A fragment of the resulting theory graph is shown in Figure 3. Here the conditional inclusion from our running example would appear as an edge from `Field` to `ResidueClassRing/EuclideanDomain`, which our theory graph layouting algorithm currently places it outside of the screenshot. We have also generated many smaller graphs representing individual parts of the library.We are able to release the generated `.axy` and OMDoc files and theory graphs. These are available at https://gl.mathhub.info/aldor/distribution and can be regenerated by running MMT.

Conclusions and Future Work

We have seen that the algebraic framework of the Aldor language and libraries carries over in a natural way to MMT theories. Certain language features, including conditional categories and *ex post facto* extensions appear to be useful more generally and we anticipate incorporating these in MMT.

References

1. M. Bronstein. SUM-IT: A strongly-typed embeddable computer algebra library. In *Proceedings of DISCO'96*, pages 22–33. Springer LNCS 1128, 1997.
2. W. Farmer, J. Guttman, and F. Thayer. Little theories. In *Automated Deduction—CADE-17*, pages 115–131. Springer, LNCS 607, August 2000.
3. R.D. Jenks and B.M. Trager. A language for computational algebra. *ACM SIGPLAN Notices*, 16(11):22–29, November 1981.
4. M. Kohlhase and F. Rabe. Experiences from exporting major proof assistant libraries. *Journal of Automated Reasoning*, 65(8):1265–1298, 2021.
5. D. Müller, F. Rabe, and M. Kohlhase. Theories as types. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *Automated Reasoning*, pages 575–590. Springer, 2018.
6. F. Rabe and M. Kohlhase. A scalable module system. *Information and Computation*, 230(1):1–54, 2013.
7. S.M. Watt. *Handbook of Computer Algebra*, chapter 4.1.3 Aldor, pages 265–270. Springer Verlag, 2003.
8. S.M. Watt. Post facto type extension for mathematical programming. In *Proc. Domain-Specific Aspect Languages*, pages 26–31. SIGPLAN, ACM, October 2006.
9. S.M. Watt, P.A. Broadbery, S.S. Dooley, P. Iglio, J.M. Steinbach, and R.S. Sutor. A first report on the $A^\sharp$ compiler. In *Proc. ISSAC*, pages 25–31. ACM, July 1994.