ICFP 2008: The 13th ACM SIGPLAN International Conference on Functional Programming
Victoria, British Columbia, Canada

Poster Session

Sunday, September 21, 2008

SASyLF: An Educational Proof Assistant for Language Theory 2
Jonathan Aldrich, Robert J. Simmons and Key Shin

A Type System for Certified Garbage Collection of Haskell Programs 4
Tim Chevalier and Andrew Tolmach

TreeView: Interactive Large Tree Visualization in Haskell 6
Jefferson Heard and Xiaojun Guan

An ML-Like Universal Module System 8
Hyeonseung Im and Sungwoo Park

Directing JavaScript with Arrows 10
Khoo Yit Phang, Michael Hicks, Jeffrey S. Foster and Vibha Sazawal

Monadic Programming Through Program Extraction 12
Josef Pohl

Unified Type Checking for Type Classes and Type Families 14
Tom Schrijvers and Martin Sulzmann

Normalization in the Dual Calculus with Sigma Reductions 16
Jeffrey Vaughan, Steve Zdancewic and Stephanie Weirich

Translucent Abstraction—Algebraic Datatypes with Safe Views 18
Meng Wang and Jeremy Gibbons

Schema Mapping Polymorphism 20
Ryan Wisnesky

Technical Report 640, Department of Computer Science, Rutgers University, Piscataway, NJ 08854, USA

SASyLF: An Educational Proof Assistant for Language Theory

Jonathan Aldrich Robert J. Simmons
Carnegie Mellon University
{aldrich, rjsimmon}@cs.cmu.edu

Key Shin
Microsoft Corporation
kshin@microsoft.com

Motivation. Teaching formal language theory is hard. By formal language theory, we mean formalizing a language through operational semantics and typing rules, and then proving meta-theorems such as type soundness. Proofs are hard in two ways: the concepts are complex and easy to get wrong, and it is hard to state things formally and precisely without making small syntactic errors (which nevertheless can hide a mistake in a proof). As a result, many published proofs have small errors or inconsistencies. Just as we have automated support for typechecking programs to make sure they do not go wrong, it would be nice to be able to check a proof to make sure it is correct.

The challenge of formal language theory is doubly hard in a teaching context. To learn effectively, students must focus on higher-level concepts like induction, yet many mistakes are made at a much lower level, e.g. skipping a step in a proof or applying an inference rule when the facts used do not match the rule's premises. Students may not even recognize they have made a mistake, and so do not seek out help. They only learn of their mistake a week or two later, when the TA hands back a paper splashed with red ink. At that point, the student may have forgotten why the mistake was made, and the learning opportunity is lost. A tool that could provide *immediate* feedback would help students get it right in the first place, and if nothing else help them know when they need to ask an instructor for help with the more challenging concepts.

Proof assistants like Isabelle/HOL [4], Coq [2], and Twelf [5] have been used to formalize language semantics and prove meta-theorems. However, even in the research community, mechanically checked proofs are the exception rather than the rule. This may be partly a productivity issue, but the steep learning curve of these tools, and the non-trivial techniques for encoding program semantics in them, likely plays a role. The OTT tool [6] allows users to write down language syntax and semantics in a convenient notation, but does not allow them to express or prove theorems—this must be done in another tool. Unfortunately, the use of these assistants in teaching formal language theory is very rare, for the same reasons, despite the help they could in theory be to students.

The SASyLF Proof Assistant In this poster, we present the SASyLF ("Sassy Elf") theorem proving assistant. SASyLF has a simple design philosophy: languages, their semantics, and their meta-theory should be written as close as possible to the way it is done on paper. Proofs are very explicit, for the benefit of teaching. Error messages are given in terms of the source proof, not in terms of the solver's underlying theory. Finally, SASyLF is specialized for reasoning about languages, programs, and logics—more generally, anything with variable binding. Variable binding is a source of trouble in other systems because it must be encoded. SASyLF is based on the logical framework LF but is restricted to the second-order case for usability: it is a Second-order Abstract Syntax Logical Framework, so it builds variable binding conventions directly into the theorem-proving language. As a result, proofs are clean, short, and look almost exactly like they do on paper.

```
syntax

e ::= fn x : tau => e[x]
  |   x
  |   e e

tau ::= unit
    |   tau -> tau

Gamma ::= *
      |   Gamma, x : tau

judgment step: e -> e

e1 -> e1'
--------------- c-app-l
e1 e2 -> e1' e2

...

judgment has-type: Gamma |- e : tau
assumes Gamma

Gamma |- e1 : tau' -> tau
Gamma |- e2 : tau'
----------------------- t-app
Gamma |- e1 e2 : tau

...
```

Figure 1. The λ-calculus in SASyLF

The SASyLF Language. We briefly show how the SASyLF language can be used to define and reason about the simply typed λ-calculus. Figure 1 shows excerpts of the definition of the simply-typed lambda calculus. We use BNF form to describe the syntax. The notation `e[x]` denotes that `x` is a variable that is bound in `e`.

```
theorem preservation: forall dt: * |- e : tau
                      forall ds: e -> e'
                      exists * |- e' : tau.

dt' : * |- e' : tau                  by induction on ds :
  case rule
    d1 : e1 -> e1'
    -------------------- c-app-l
    d2 : e1 e2 -> e1' e2
  is
    dt' : * |- e' : tau          by case analysis on dt :
      case rule
        d3 : * |- e1 : tau' -> tau
        d4 : * |- e2 : tau'
        ------------------------- t-app
        d5 : * |- (e1 e2) : tau
      is
        d6 : * |- e1' : tau' -> tau
                        by induction hypothesis on d3, d1
        dt' : * |- e1' e2 : tau   by rule t-app on d6, d4
      end case
    end case analysis
  end case

  case rule... // other rules not shown
end induction
end theorem
```

Figure 2. Preservation proof for the λ-calculus in SASyLF

The operational semantics of the λ-calculus are defined by giving the form of the judgment first and then a series of inference rules. Each rule has one or more premises, one per line, followed by the rule's name and the conclusion.

The `has-type` judgment is similar to the reduction judgment, but here the **`assumes`** declaration tells SASyLF that `Gamma` is not merely a syntactic form, but is a list of assumptions in a hypothetical judgment form.

Figure 2 shows one case in the proof of type preservation for the simply-typed λ-calculus in SASyLF (one case is elided). Like Twelf, SASyLF supports theorems of the form "for all *≪list of metavariables and judgments≫* there exists *≪judgment≫*."

Syntactically, one must give a name for the theorem and for the derivation of each input judgment. The derivation names (`dt` and `ds`) are used to refer to judgments within the proof.

A proof is a list of judgments, each with a justification. The preservation theorem uses induction, induction hypothesis, case analysis, application of inference rules, and substitution as justifications.

The preservation proof begins by stating the judgment we want to prove, `* |- e' : tau`, giving it the name `dt'`, and stating that it is justified by induction over the derivation of the evaluation judgment `ds`.

We immediately do a case analysis on the rules used to derive this judgment. Each case in the case analysis is introduced with one of the rules that could be used to generate `ds`. The rule is stated using fresh metavariables that are bound in the body of the case (`e1`, `e2`, and `e1'` in the case of `c-app-l`). SASyLF matches the conclusion of the rule to the judgment we are case-analyzing, and determines that `e` has been substituted with `e1 e2` and that `e'` has been substituted with `e1' e2`.

We proceed to further case analyze on the typing derivation `dt`. Since we know that `e=e1 e2` there is only one possible case, rule `t-app`. SASyLF will try all the other rules but will discover that their conclusions don't match the form `e1 e2`; if any of them matched, SASyLF would report an error stating which rule needs to be added to the case analysis.

In these two nested cases, we have learned that `e1 -> e1'` and `* |- e1 : tau' -> tau`. We therefore can apply the induction hypothesis, naming the two facts just mentioned with their names `d1` and `d3`. SASyLF checks that the derivations used to instantiate the "forall" clauses of the theorem in fact match those clauses, and checks that the resulting derivation `d6 : * |- e1' : tau' -> tau` is in fact what you get from applying the theorem to those inputs. SASyLF also verifies that the derivation passed in for the argument of the theorem we are doing induction over, `e1 -> e1'`, is a subderivation of the thing we analyzed by induction, `e -> e'`.

Finally, SASyLF checks that the last step in the proof of each case (and of the main theorem) is a statement of the thing we are trying to prove, namely `* |- e' : tau` (where in this case `e'=e1' e2`). We get this by applying rule `t-app` to the derivations we got from the second case analysis and the induction hypothesis. SASyLF performs checks similar to those for the induction hypothesis, except of course that the subderivation check is not relevant.

Experience. We have used SASyLF for one relatively simple assignment in the first author's Spring 2008 course on program analysis. While students found some usability problems with the tool (many since corrected), there were preliminary indications that the tool aided in learning processes and that users who were sucessful with the tool would use it again. More details of our experience are described in Functional and Declarative Programming in Education 2008 [1].

Implementation. An information presentation in the 2008 Workshop on Mechanizing Metatheory will discuss the principles behind the SASyLF prover, including its basis in the logical framework LF [3]. An implementation of the tool is available along with additional information at **`http://www.sasylf.org/`**

References

[1] J. Aldrich, R. J. Simmons, and K. Shin. SASyLF: An Educational Proof Assistant for Language Theory. In *Functional and Declarative Programming in Education*, Sept. 2008.

[2] Y. Bertot and P. Castran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.

[3] R. Harper, F. Honsell, and G. D. Plotkin. A Framework for Defining Logics. *J. ACM*, 40(1):143–184, 1993.

[4] T. Nipkow, L. C. Paulson, and M. Wenzel. Isabelle/HOL: A Proof Assistant For Higher-Order Logic. *Lecture Notes in Computer Science*, 2283, 2002.

[5] F. Pfenning and C. Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *International Conference on Automated Deduction*, pages 202–206, July 1999.

[6] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. Ott: Effective Tool Support for the Working Semanticist. In *International Conference on Functional Programming*, pages 1–12, 2007.

A Type System for Certified Garbage Collection of Haskell Programs

Tim Chevalier Andrew Tolmach
Portland State University
{tjc,apt}@cs.pdx.edu

Abstract

We present a simple type system to ensure the safety of a precise garbage collector, and base on it three intermediate languages in a type-preserving compilation pipeline from Haskell to assembly language. The safety guarantees come at the cost of dynamic checks, and through experiment, we are determining the nature of the trade-off between safety and efficiency in this particular instance.

1. Safe Haskell compilation

Haskell's static type system helps programmers find and fix a large class of program errors at compile time, preventing dangerous or at least embarrassing errors from occurring in production code. But every Haskell program that is compiled by the Glasgow Haskell Compiler (GHC) yields an executable that is linked with code from the runtime system (RTS). GHC's RTS consists of over 30,000 lines of C code. Automated verification of C programs on such a scale is beyond the scope of current technology. Thus, the RTS is a weak link in any argument about the safety of Haskell compilation systems. We are removing this weak link by developing an RTS for Haskell that provides only the services that every program uses. Such an RTS would be smaller and thus more amenable to formal proofs than GHC's RTS. The minimal RTS will still have to include a garbage collector, and so in this poster we explore how to compile Haskell through typed intermediate languages that facilitate verifying the collector.

1.1 Using CompCert

Rather than building a new compiler back-end and RTS from scratch, we choose to appropriate the back-end of an existing compiler, CompCert. CompCert is a certified compiler for a subset of C (Leroy 2006). When we say that it is "certified", we mean that it compiles source code to assembly code in a way that has been formally proven to preserve the source code's semantics. Thus, using CompCert limits our proof responsibilities to type preservation for the path from Core to Cminor. Note that our proofs assume nothing about the origin of Core code, so we need prove nothing about the front-end translation from Haskell to Core.

1.2 From GHC to CompCert

CompCert's back-end compiles code in the intermediate language Cminor (Leroy 2006) to machine code, so to use the back-end, we must compile Haskell to Cminor. Fortunately, GHC's front-end already does much of the work for us. GHC desugars Haskell into the much simpler Core intermediate language, and GHC's "External Core" feature (Tolmach et al. 2008) lets us divert this intermediate representation to a text file. However, there are still a number of challenges in compiling Core to Cminor. Figure 1 summarizes those challenges.

	Core	Cminor
Evaluation	call-by-need	call-by-value
Closures	yes	no
Types	polymorphism, type constructors, recursive types, type-safe casts, primitive unboxed types	integers, floats
Control flow	case matching on algebraic data types with data constructors	conditionals

Figure 1. Differences between Core and Cminor

To address the differences between Core and Cminor in a modular way, we introduce three new intermediate languages between Core and Cminor, with corresponding type-preserving translations (or, in the case of the final two translations, semantics-preserving translations.) Figure 2 shows the phases of the compilation pipeline. (The black boxes denote stages we have built, whereas the other boxes denote existing components we have used.) The compiler translates Core to E♭, a strict language with explicit laziness and with a very simple type system. It then translates E♭ to D, which is simpler than E♭ in that it lacks first-class functions, but which has a slightly more complicated type system. It translates D to GCminor, an implicitly garbage-collected variant on Cminor, and finally compiles GCminor to Cminor.

1.3 Garbage collector correctness

We choose to focus on garbage collector correctness, as it is a well-defined problem that is crucial to the correctness of the entire RTS. Any implementation of a type-safe language must incorporate a garbage collector, and collector bugs are not unheard-of (McCreight et al. 2007). Buggy collectors cause memory leaks, or worse, dangling pointers that result in inappropriate memory accesses when dereferenced—in short, the same bugs that bedevil the devotees of languages that have manual memory management.

Verifying the collector requires verifying that compiled programs respect the contract between mutator and collector, and so we further focus on that particular property. Also, we target precise garbage collectors that must be able to statically distinguish pointers from non-pointers. Thus, we consider it a prerequisite for collector correctness that the compiler's intermediate language allows pointers to be distinguished statically from non-pointers. In the remainder of this summary, we present a type system for doing exactly that, and a series of intermediate languages designed around that type system. Note that we limit our scope to proving safety for the garbage-collected system (similarly to most related work), rather than proving semantic preservation for the entire compiler.

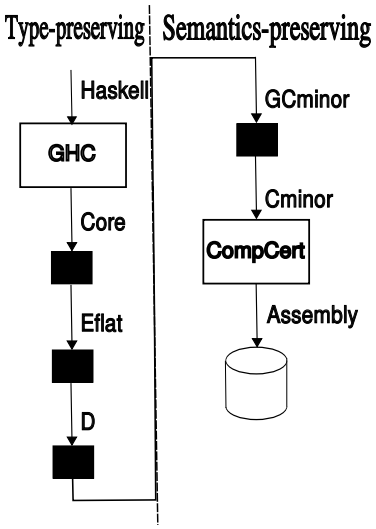


Figure 2. Phases of the compilation pipeline

2. Type system

We require a very simple type system: one that distinguishes pointers from non-pointers and does nothing else. But our compilation pipeline starts with Core, a language with a complex type system. It is difficult to isolate just a few features of Core's type system to remove, as they interact in unexpected ways. So we had to either: maintain Core's rich type system all the way down the compilation pipeline, necessitating much machinery that is irrelevant to our goal; or replace Core's type system with an extremely simple type system. We chose the second option. A simple type system makes it easier to prove that each compilation phase is type-preserving, which in turn obviates the need to prove the entire compiler correct. However, the cost of this simplicity may be very high: as we explain shortly, we pay for simplicity in runtime checks.

The key idea is to collapse all types in Core whose values are represented by a pointer to garbage-collected memory at runtime—that is, *boxed* types—into a single type, written as □. □ encompasses Core's function types, polymorphic types, type constructors, type constructor applications, and type variables. Operationally, if a value has the □ type, that means that it is a pointer that can be dereferenced at runtime to obtain both a data object, and a header that provides more precise type information (e.g., whether the object is a function closure or a data structure).

Our first intermediate language, E♭, has only three types: □, along with two of Core's primitive unboxed types `Int#` and `Float#`. The translation from E♭ to D replaces first-class functions with explicit closures. Thus, D's type system keeps the three types from E♭ and adds record types to type these closures.

We think we have identified the simplest system that statically distinguishes pointers from non-pointers, and this system is useful for proving the safety of a compilation system including precise garbage collection while avoiding unrelated concerns. But our design has performance implications:

- Every function application potentially entails a runtime check that the operator is actually a function with the appropriate argument type. This means that function closures must be tagged with their expected argument types at runtime. The checks are necessary because all functions (and constructed values) have static type □. Thus, the typechecker for E♭ cannot verify that the operator in every application is a function of the appropriate type, and soundness necessitates runtime checks.
- Every *case* expression potentially entails a runtime check that some alternative matches the scrutinee. In Core, *case* expressions are exhaustive, because each datatype has a (usually) small set of constructors, and GHC automatically inserts explicit exception-throwing statements when programmer-written *case*s are non-exhaustive. But in E♭, all data constructors in the entire program are effectively constructors of a single algebraic datatype. Thus, there is no type information to provide a static guarantee that *case*s inspect values of the appropriate type. However, the type system *does* provide a static guarantee that *case* expressions only scrutinize values of type □.

We say "potentially" because some of these checks can potentially be eliminated by optimizations (for example, checks can be omitted for calls to known functions), but not all of them will be, and type tags will inevitably increase space usage.

Though the story begins to seem rather grim, the bright side is that this simple type system makes it easy to prove type preservation for the translation steps from Core to E♭ and from E♭ to D. We have proven type preservation for the Core-to-E♭ and E♭-to-D translations. As we mentioned earlier, our goal is to prove safety rather than full semantic preservation, so proving type preservation suffices. However, GCminor and Cminor lack appropriately rich type systems to admit type preservation proofs for the last two translation phases. So, we will prove semantic preservation for the last two phases in order to complete the safety proof for the whole pipeline.

3. Status

This poster describes work in progress. So far, we have implemented the Core-to-E♭ and E♭-to-D translations and are implementing the D-to-GCminor translation. Once the implementation is complete, we will be able to estimate the runtime cost of the dynamic type checks our system introduces by compiling and running benchmark programs. This will guide us in developing appropriate compiler optimizations to reduce the number and cost of these checks, which in turn will help answer the question of whether the resulting safety guarantees justify the cost of the checks. We think it is worthwhile to explore the consequences of exchanging some efficiency for a simpler correctness argument.

References

X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In 33rd Symposium on Principles of Programming Languages, pages 42–54. ACM Press, 2006.

A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying garbage collectors and their mutators. In PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 468–479, New York, NY, USA, 2007. ACM.

A. Tolmach, T. Chevalier, and the GHC Team. An external representation for the GHC Core language, June 2008. URL `http://cs.pdx.edu/~tjc/ext-core.pdf`.

TreeView: Interactive Large Tree Visualization in Haskell

Jefferson Heard
Renaissance Computing Institute
University of North Carolina
jeff@renci.org

Xiaojun Guan
Renaissance Computing Institute
University of North Carolina
xguan@renci.org

Abstract

We present TreeView and give an example application of TreeView called ProteinViz. Developed entirely in Haskell, ProteinViz is an interactive visualization for trees built while clustering genomics and proteomics data. TreeView is the library that we developed as part of the application and later generalized. TreeView and ProteinViz take advantage of many of the features of Haskell and give an example of how type-classes, laziness, and recursive data-types integrate well into a real-world application.

Categories and Subject Descriptors D3.3 [**Language Constructs and Features**]: Modules, packages

General Terms: Design, Human Factors.

Keywords: haskell, FP, visualization, infoviz, tree view

1. Introduction

As a functional programming language with full support for native-code compilation, OpenGL 2.1, laziness, recursive data types, and type classes, Haskell is the perfect language for building a framework to visualize large tree data structures. What started as a visualization of a protein tree with 56,000+ nodes has turned into a general framework for declaring trees visualizable and displaying them in two and three dimensions.

We present TreeView and ProteinViz. ProteinViz is a tool for viewing hierarchical clusterings of genomic and proteomic data. TreeView is the Haskell library that we developed to support ProteinViz and other large tree viewing applications.TreeView handles binary and general trees, and takes advantage of laziness to only load exactly the amount of data needed to display what the user wants to view on the screen. This both speeds up application load times and allows us to run applications using the library on laptop hardware and older desktops with less memory/CPU.

There are tree viewers for genomic data now, such as Java TreeView [1], and Heirarchical Cluster Explorer [2]. There programs cannot, however, handle trees of tens of thousands of nodes very well. Leaf nodes become too densely packed to distinguish. Interactivity is diminished. Collapse/Expand is not supported, and much screen space is wasted on data that is not relevant to our applications.

2. TreeView

TreeView defines three type classes: VisTree, TextTree, and IndexableTree, which define the operations a tree data type must support for TreeView to generate geometry, perform node search, and provide detailed descriptions of each node. A datatype that implements these classes can then use the methods in the module Geometry to lazily load data as well as create and render 2D and 3D OpenGL representations of these trees.

To make a binary tree into a visualizable tree, one needs to implement the 6 methods in Graphics.Visualization.Tree.VizTree:

- `left :: a -> a` – the left child.
- `right :: a -> a` – the right child.
- `leaf :: a -> Bool` – whether or not this is a leaf.
- `value -> a -> Float` – a numeric value determines the color of the node on screen.
- `text :: a -> String` – A brief summary string.
- `detail :: a -> String` – A more detailed string.

To make a general tree into a visualizable tree, one implements the method *children* instead of *left* and *right*. Now to create geometry from a tree, the user can simply import Graphics.Visualization and call either of:

- `generatePolarGeometry pr . index $ tree`
- `generateCartesianGeometry pr . index $ tree`

where pr is a Parameters object containing a min and max value for the value function, a colormap function, a gamma value for the colormap, a radius (or height) for the rendered tree, and the total number of leaves in the tree. These function calls create the geometry for the tree, and this tree can the be passed to the functions in Graphics.Visualization.Tree.RenderGeometry to actually make OpenGL calls to put the object on screen. Even the generation of geometry is purely-functional, reserving all calls that result in bringing the computation into the IO monad for the actual process of rendering the geometry to the screen. This allows a programmer to transform the geometry through series of purely-functional filters that can perform things like force-directed layout and transformations to fit the overall scene geometry.

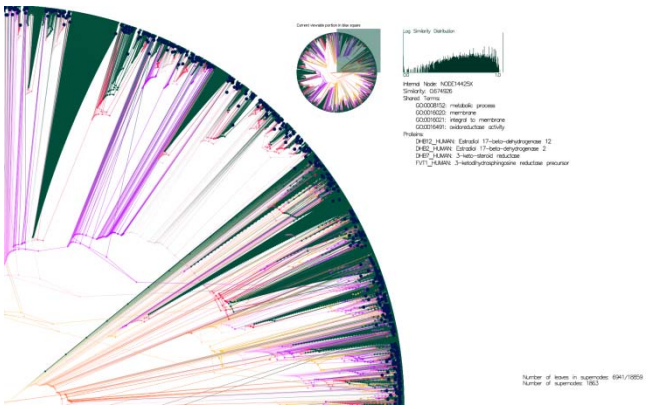


Figure 1. ProteinViz over 58,000 nodes. Leaves correspond to individual human proteins

3. ProteinViz

ProteinViz was developed in collaboration with Dr. William Kaufmann of UNC Chapel Hill to provide visual analytics aid in an ongoing mapping of a melanoma genome. The point is to have a visual gauge of the accuracy of the gene ontology terms assigned to a protein or group of proteins by putting proteins in the context of an entire genome. ProteinViz uses the Haskell TreeView code to display 58,000 nodes of a human genome tree in a radial tree view in Figure 1. Colors reflect the similarity between nodes. The leaves represent individual proteins, and the internal nodes represent heirarchal clusterings of those proteins based on one of several similarity metrics. At each node of the tree, the user can click on a node and see what Gene Ontology terms are shared between all the proteins reachable from that node, and all the names and defitnitions of the proteins in that node. Also, we use the indexing function of TreeView to arrange leaves so that proteins that are similar (using our similarty metric) are grouped together along the rim.

In the user interface, the names of the proteins accessible as children of a highlighted node are listed to the right of the main body of the tree. An inset to the upper right of the main viewing portion shows the relationship between the quarter of the tree that is visible to the user and the whole tree. At the top far right is a histogram breaking down internal nodes' values.

There are various automatic approaches which can be used to group and collapse subtrees at will – these use the search functions provided by the IndexableTree type-class, and collapse portions of the indexed tree before passing it to the geometry functions. The number of nodes collapsed in this way is shown at the bottom right of the visualization as can be seen in Figure 1. Users can click internal nodes to collapse/expand subtrees and to show annotations using TreeView's RenderText module.

Users can control almost the entire visualization from the mouse, using the right button to expand and contract nodes, the scrollwheel to rotate along the tree, and the left button to highlight and navigate around the tree. Additionally, the left, right, and up keys on the keyboard allow the user to navigate from the currently highlighted node up, down-and-left, or down-and-right from the selected node in the tree.

We took advantage of the laziness inherent in Haskell to create a tree that conceptually contained much more data in memory than typical hardware we expected this to run on would hold, including the full set of shared gene ontology terms and definitions of these terms and proteins at each node. Laziness mean that the majority of this data was never loaded unless it either directly pertained to the user (i.e. it was clicked on and is highlighted) or directly pertained to the values of the six TreeView functions. The result was a treeview that came up vastly quicker than the old Java TreeView that we used previously, remained more responsive, used less memory during the session.

We have tested ProteinViz and TreeView on specialized visualization hardware and typical laptop hardware. The visualization maintains interactivity and usability even on a 15" laptop with 1GB of RAM and an older processor.

TreeView is available at http://bluheron.renci.org/TreeView

References

[1] Saldanha, A. Java Treeview – extensible visualization of microarray data. *Bioinformatics*. 20(17) 2004.

[2] Seo, J., Shneiderman, B. Interactively Exploring Heirarchical Clustering Results. *Computer* 35. pp 80-86, 2002.

[3] Sheth, N., Borner, K., Baumgartner, J., Mane, K., and Wernert, E. Treemap, Radial Tree, and 3D Tree Visualizations. *IEEE Visualization Conference*, 28-129. 2003.

An ML-like universal module system

Hyeonseung Im and Sungwoo Park

Department of Computer Science and Engineering
Pohang University of Science and Technology
Gyeongbuk, Republic of Korea
{genilhs,gla}@postech.ac.kr

1. Motivation

Considerable effort has been devoted to the design of module systems to support modular programming. Languages with module systems consider programs as consisting of *modules*, each of which is a collection of related *declarations* (such as definitions of data structures and associated operations). The *interface* to a module specifies which components in the module are accessible from the outside. Given a module implementation and an interface, the compiler can check if the implementation conforms to the interface specification. This allows programmers to develop their own module using only the interfaces of the modules on which it depends, which is the most necessary feature of modular programming.

The ML module system [8], among others, has been extensively studied due to its powerful support for modular programming and data abstraction. It has two linguistic levels: a *core language* for defining module components and a *module language* for providing modular programming constructs. The module language consists of *structures* and *functors* which constitute modules, and *signatures* and *functor signatures* which constitute module interfaces. A structure combines related declarations with a consistent naming scheme. Functors are used to express parameterized modules (*i.e.,* functions from modules to modules). Signatures and functor signatures are interfaces for structures and functors, respectively.

Although the ML module system provides powerful support for modular programming, it is not easy to use for other languages. The reason is that its module language is closely connected to its core language (*i.e.,* Core ML). For example, consider the most popular dialects of ML: Standard ML (SML) [10] and Objective Caml (OCaml) [1]. In both SML and OCaml, core language constructs (such as variables, types, and expressions) are exposed at the module language level, and types play a key role in connecting the module language to the core language. Let us give a specific example for each language:

- The static semantics for the SML module language heavily depends on the static semantics for its core language especially due to its use of semantic object *type names* which represent type identity at the semantic level. That is, type names must be incorporated into the core language.
- The static semantics for the OCaml module language only assumes the existence of the static semantics for the core language. Still, however, the OCaml module system requires that the core language 1) provide a conversion operation from access paths to types to support *type strengthening*, and 2) define a type matching relation between access paths [7].

Furthermore, when the core language is extended or modified. both SML and OCaml module languages may need to be extended or modified accordingly.

In order to provide other languages with modular programming facilities of the ML module system, we propose a new ML-like universal module system.

- What we mean by "ML-like" is that the module language provides usual modular programming constructs in ML, such as structures, signatures, and functors.
- What we mean by "universal" is that the module language is designed to be as independent of the core language as possible; we put minimal requirements on the core languages that practically any languages can satisfy. Furthermore, by minimizing dependencies between the core and module languages, the extension or modification of the core language does not lead to that of the module language, and vice versa.

We first formulate the requirements on the core language. We then define the static and dynamic semantics for the module language, and show the correctness of the module system in terms of usual progress and preservation properties.

2. Core language requirements

At the syntactic level, the core language needs to provide two constructs $cdec$ and $cspec$ that denote core language declarations and specification respectively (*e.g.,* type declarations and specifications in ML). Their actual definition is not exposed to the module language, thus preventing it from relying on the specific core language features. At the semantic level, the core language needs to provide a decidable procedure for checking $cdec$s against $cspec$s and inferring the $cspec$ of a $cdec$ in order to define the static semantics for the module language; it need not necessarily be a type-checker.

With the introduction of the module system, a $cdec$ ($cspec$) may refer to $cdec$s ($cspec$s) that are defined in other modules; we assume that all those references are allowed only via *paths* which are sequences of module names. Paths are abstracted out at the core language level, and thus the module language may change the actual implementation of paths without affecting the core language. They are the only components of the module language that are visible to the core language.

Last but not least, the core language needs to provide a *path substitution operation*. Without it, the module language cannot replace paths in $cdec$s with other paths. Provided that a $cspec$ is the specification of a $cdec$, the application of path substitution to both $cspec$ and $cdec$ must preserve the relationship between them. The path substitution operation plays a key role in defining the static and dynamic semantics for the module language (Section 4).

3. Path resolution problem

We define the static semantics for the module language as a procedure for checking modules against interfaces and inferring the

```
module X = struct
  type t = int
  module F (A : sig  end) = struct
    type s = t
    module B = struct type v = s * s end
  end
end
type u = X.t
module R = X.F (struct  end)
```

Figure 1. Example code

interface of a module on the assumption that the core language provides a procedure for checking *cdec*s against *cspec*s and inferring the *cspec* of a *cdec*. The static semantics is decidable if the procedure of the core language is decidable. We define the dynamic semantics for the module language as defunctorization that eliminates all functor applications at compile-time. Elsman's interpretation of modules [4] also performs defunctorization, but it relies on the static interpretation of the underlying core language. Our dynamic semantics differs from his work in that it is defined only in terms of modules.

In order to define the static and dynamic semantics, we need to solve a *path resolution problem*. The path resolution problem arises because the point that a module is defined differs from the point that the module is used; paths valid at a certain point may not be valid at other points. Consider the example code in Figure 1 written in OCaml. To simplify the discussion, we consider type declarations (*e.g.,* `type t = int`) as one class of *cdec*s, and assume that the core language uses the standard dot notation for module access. When functor `F` is defined, type `t` declared in `type t = int` is accessed as `t`. But, at the functor application point, `t` becomes invalid; it should be accessed as `X.t`.

The path resolution problem is solved by a simple substitution-based approach [6, 11, 7] if it is known to the module language 1) how *cdec*s and *cspec*s use paths, and 2) how the core language semantics interprets them. For example, when type-checking functor application `X.F (struct end)`, it first generates substitutions for components defined in module `X` (*i.e.,* $\{\texttt{t}\rightarrow\texttt{X.t}\}$); then it applies the substitution to `F`'s signature, thus solving the problem. (*i.e.,* `F`'s signature `functor (A : sig end) -> sig type s = t ... end` is then converted to `functor (A : sig end) -> sig type s = X.t ... end`.) However, since both conditions 1) and 2) are not met in our module system, we take a different approach to solve the path resolution problem.

4. Semantic paths and path closure conversion

In order to solve the path resolution problem, we take an approach that converts relative paths to *absolute paths*. Put simply, absolute paths are paths that are valid anywhere. To provide the concept of absolute paths, we introduce the notion of *semantic paths* for the static semantics and *fully specified paths* for the dynamic semantics.

Semantic paths are similar to *structure stamps* in [9]. Each structure has a unique semantic path and a semantic path uniquely determines its corresponding structure. Due to the introduction of semantic paths, we define our static semantics as an elaboration of syntactic objects into semantic objects as in [10]. To solve the path resolution problem arising in the static semantics, we translate syntactic paths into the corresponding semantic paths during the elaboration process. (The translation uses the path substitution operation provided by the core language.) For example, in Figure 1, if α is the semantic path of module `X`, declaration `type s = t` elaborates to `type s =` α`.t` and `type u = X.t` to `type u =` α`.t`.

Fully specified paths are paths that start from predefined top module name `T`. A fully specified path uniquely determines its corresponding module as does a semantic path. (We require that all modules declared at the same nesting level have distinct names.) We solve the path resolution problem arising in the dynamic semantics by converting syntactic paths to the corresponding fully specified paths. (The conversion uses the path substitution operation provided by the core language.) We call this conversion a *path closure conversion*.

However, we cannot fully specify paths used in declarations inside submodules defined in the functor body (*e.g.,* `type v = s * s end` in Figure 1). Therefore, we define the path closure conversion in such a way that functors are parameterized not only by their arguments but also by names that will be bound to their application results. This allows functor body to be fully specified. For example, assuming that variable `Z` ranges over fully specified paths, module `F` in Figure 1 is converted as follows:

```
module F (A : sig  end, Z) = struct
  type s = T.X.t
  module B = struct type v = Z.s * Z.s end
end
```

The path closure conversion preserves the typing property of a module. Let `S` be the signature of module `X`, and $\texttt{S}^\circ$ and $\texttt{X}^\circ$ be the conversion results; then $\texttt{S}^\circ$ is the signature of module $\texttt{X}^\circ$.

With the aid of the path closure conversion, defunctorization is defined as follows. When eliminating a functor application, we replace the formal arguments of the functor with its actual arguments, and path variable `Z` with the actual fully specified path that will be bound to the application result. The safety of defunctorization is defined in terms of usual progress and preservation properties.

5. Conclusion

For any languages that satisfy the requirements presented in Section 2, our module system provides modular programming constructs in ML, such as structures, signatures, and higher-order functors. However, since we do not assume that the core language provides type declarations, it is hard to investigate the interaction between type abstraction and our module system [5, 6], and thus recursive modules [3]. We plan to support type abstraction and recursive modules in our framework.

References

[1] Objective Caml. `http://caml.inria.fr`.

[2] Davide Ancona and Elena Zucca. A calculus of module systems. *Journal of Functional Programming*, 12(2):91–132, 2002.

[3] Derek Dreyer. A type system for recursive modules. In *ICFP '07*.

[4] Martin Elsman. Static interpretation of modules. In *ICFP '99*.

[5] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL '94*.

[6] Xavier Leroy. Manifest types, modules, and separate compilation. In *POPL '94*.

[7] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[8] David MacQueen. Modules for Standard ML. In *LFP '84*.

[9] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In *ESOP '94*.

[10] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of Standard ML (revised)*. The MIT Press, 1997.

[11] Zhong Shao. Transparent modules with fully syntactic signatures. In *ICFP '99*.

Directing JavaScript with Arrows

Khoo Yit Phang Michael Hicks Jeffrey S. Foster Vibha Sazawal
University of Maryland, College Park
{khooyp,mwh,jfoster,vibha}@cs.umd.edu

Abstract

Event-driven programming in JavaScript often leads to code that is messy and hard to maintain. We have found *arrows*, a generalization of *monads*, to be an elegant solution to this problem. Our arrow-based *Arrowlets* library makes it easy to compose event-driven programs in modular units of code. In particular, we show how to implement *drag-and-drop* modularly using arrows.

1. Event-driven JavaScript is Messy

JavaScript is the *lingua franca* of Web 2.0, and is the basis of highly interactive web applications such as Google Maps and Flickr. Because JavaScript code runs in a client-side web browser, applications can present a rich, responsive interface without the latency associated with client-server communication.

JavaScript, however, is a single-threaded language that has to cooperate with the web browser's user interface. The JavaScript API is designed in event-driven style, and it is crucial that event callbacks execute quickly so that new events are handled in a timely fashion. Long-running loops, animations, and state machines are typically implemented by chaining callbacks, each of which ends by registering one or more additional callbacks. Unfortunately, writing this style of code is typically tedious, error-prone, and non-modular because the callback chaining code (the "plumbing") is often hard-coded and strewn throughout the program.

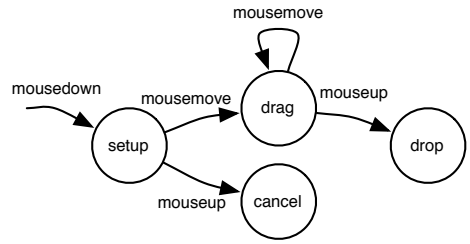
Drag-and-drop is a prototypical example that illustrates these issues. Figure 1 shows a common way to implement drag-and-drop in JavaScript. The four states—setup, drag, drop and cancel—are implemented as event handlers, and each handler is responsible for installing handlers for the next states. For example, when setup executes, it has to disable itself (line 4) and install drag and cancel (lines 5–6). Since the states are hard-coded, we cannot re-use setup in another application, and if we want to insert a new state in the state machine, we may need to edit several different handlers.

2. Arrows Point the Way

Inspired by libraries such as Fudgets (Carlsson and Hallgren 1993) and Yampa (Hudak et al. 2003) in Haskell, we discovered that *arrows* (Hughes 2000), a generalization of *monads*, is an elegant way to compose event-driven programs in JavaScript.

Arrows support at least two operations: *arr f* lifts a function *f* into an arrow, and $f \ggg g$ composes a new arrow where *g* is applied to the output of *f*. Figure 2 shows two (very) simplified definitions of function arrows. In Haskell, we define function arrows as the *Arrow* ($\rightarrow$) type with *arr* as the identity function and $\ggg$ as function composition. In JavaScript, we extend every function object with the arrow interface by adding the methods A and next, equivalent to *arr* and $\ggg$, to the built-in Function.prototype object.

We can apply arrows to the observation that event listener functions such as addEventListener are *continuation passing style* (CPS) functions where the callbacks are the continuations. By abstracting




```
1  function setup(event) { /* likewise for drag, drop, cancel */
2      var target = event.currentTarget;
3      /* setup drag-and-drop */
4      target.removeEventListener("mousedown", setup, false);
5      target.addEventListener("mousemove", drag, false);
6      target.addEventListener("mouseup", cancel, false);
7  }
8
9  document.getElementById("dragtarget")
10         .addEventListener("mousedown", setup, false);
```

Figure 1: Drag-and-drop state diagram and JavaScript code

```
1  instance Arrow (→) where
2      arr f       = f
3      (f ⋙ g) x = g (f x)
4
5  add1 x = x + 1
6  add2 = add1 ⋙ add1
7  result = add2 1      {- returns 3 -}
```

```
1  Function.prototype.A = function() { /* arr */
2      return this;
3  }
4  Function.prototype.next = function(g) { /* ⋙ */
5      var f = this; g = g.A(); /* ensure g is a function */
6      return function(x) { return g(f(x)); }
7  }
8
9  function add1(x) { return x + 1; }
10 var add2 = add1.next(add1);
11 var result = add2(1);      /* returns 3 */
```

Figure 2: Function arrows in Haskell (top) and JavaScript (bottom)

event listeners into a library of CPS arrows, we can still write event handlers as regular functions, and use arrow operations to lift and compose handlers with listeners. This nicely encapsulates the event handling code in arrows, and the plumbing in arrow combinators, thereby separating them and making re-use practical.

3. *Arrowlets*

Following this inspiration, we developed *Arrowlets*, a JavaScript library for event-driven programming. The core building block of the Arrowlets library is the AsyncA arrow prototype, from which all arrows are built. A simplified version of AsyncA is shown in Figure 3. The AsyncA constructor (lines 2–4) creates an arrow

```
1  /* AsyncA is the prototype for asynchronous arrows */
2  function AsyncA(cps) {                    /* constructor */
3      this.cps = cps; /* cps :: (x, k) → () */
4  }
5  AsyncA.prototype.AsyncA = function() {   /* identity */
6      return this;
7  }
8  AsyncA.prototype.next = function(g) {    /* sequencing */
9      var f = this; g = g.AsyncA();
10     /* CPS function composition */
11     return new AsyncA(function(x, k) {
12         f.cps(x, function(y) { g.cps(y, k); });
13     });
14 }
15 AsyncA.prototype.run = function(x) {     /* running */
16     this.cps(x, function(y) {});
17 }
18 Function.prototype.AsyncA = function() { /* lifting */
19     var f = this; /* wrap f in CPS function */
20     return new AsyncA(function(x, k) { k(f(x)); });
21 }
22
23 /* An EventA arrow waits for an event to fire on a target */
24 function EventA(eventname) {             /* constructor */
25     if (!(this instanceof EventA)) return new EventA(eventname);
26     this.eventname = eventname;
27 }
28 EventA.prototype = new AsyncA(function(target, k) {
29     var f = this;
30     function handler(event) {
31         target.removeEventListener(f.eventname, handler, false);
32         k(event); }
33     target.addEventListener(f.eventname, handler, false);
34 });
```

Figure 3: Simplified AsyncA arrow prototype and EventA arrow

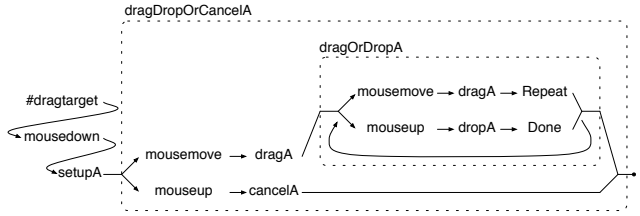
around a cps function, and arrow combinators such as next (lines 8–14) are implemented in CPS. We can execute an AsyncA arrow by invoking the run method (lines 15–17), which calls cps with an input x and an empty terminal continuation. Finally, we extend Function.prototype with an AsyncA method to lift a regular function into an AsyncA arrow (lines 18–21).

With AsyncA, we can define the EventA event listener arrow. The EventA constructor (lines 24–27) creates an arrow that listens to an event named eventname. The constructor contains a convenient JavaScript idiom (line 25) that allows us to create EventA arrows without using the **new** operator. EventA inherits from AsyncA (lines 28–34), and is built around a CPS function that listens for an event. When EventA executes, it first installs a stub event handler on the input target element. After the event fires, it uninstalls the stub event handler, and invokes the continuation k—the next arrow and actual event handler—with the received event. We chose to uninstall the event handler as this corresponds to transitions in a state machine, which is one of our motivating use cases.

Our library also provides other arrows, e.g., the ElementA arrow that ignores its input and returns a specified element from the host HTML document. We also provide additional combinators such as repeat, which puts an arrow in a loop; the arrow may return Repeat(x) to run another iteration, or Done(x) to end the loop. Another useful combinator is or, which composes two event arrows and allows only one, whichever is triggered first, to execute.

4. Drag-and-Drop with Arrowlets

Figure 4 shows how we can use Arrowlets to implement drag-and-drop in an intuitive and modular way. As before, we write four event handlers—setupA, dragA, dropA and cancelA—corresponding to the four states in drag-and-drop. Like setup in Figure 1, setupA (lines 1–4) is written as a regular function, but in contrast, it does not contain any callback plumbing code. Since it is not tied to the other handlers, it can be re-used in other applications.




```
1  function setupA(event) { /* likewise for dragA, dropA, cancelA */
2      /* setup drag−and−drop */
3      return event.currentTarget;
4  }
5
6  var dragOrDropA =
7      (    (EventA("mousemove").next(dragA).next(Repeat))
8       .or(EventA("mouseup").next(dropA).next(Done))
9      ).repeat();
10
11 var dragDropOrCancelA =
12         (EventA("mousemove").next(dragA).next(dragOrDropA))
13     .or(EventA("mouseup").next(cancelA));
14
15 var dragAndDropA = /* drag−and−drop */
16     EventA("mousedown").next(setupA).next(dragDropOrCancelA);
17 ElementA("dragtarget").next(dragAndDropA).run();
18
19 var jigsawA =          /* alternative use of dragOrDropA */
20     (nextPieceA
21         .next( (dragOrDropA.next(repeatIfWrongPlaceA)).repeat() )
22     ).repeat();
```

Figure 4: Drag-and-drop arrow diagram and code

The plumbing that composes the handlers has been extracted into the remainder of the code in Figure 4. We use various arrow combinators to compose the handlers and appropriate event listeners into the drag-and-drop state machine. We can also organize the composition modularly in three parts. For example, the first part, dragOrDropA (lines 6–9), is a repeat loop that handles the dragging animation during mousemove events, and the dropping action after a mouseup. In addition to drag-and-drop (lines 15–17), we can even re-use dragOrDropA in a jigsaw puzzle game (lines 19–22).

Finally, this drag-and-drop composition, shown graphically above Figure 4, mirrors the state diagram in Figure 1. We find it quite intuitive to convert a state diagram into an arrow composition.

In conclusion, arrows makes it easy to write event-driven programs in an intuitive and modular way. The Arrowlets library is available at our website (`http://www.cs.umd.edu/projects/PL/arrowlets`), along with a technical report, API documentation and several live examples.

Acknowledgments

This research was supported in part by National Science Foundation grants IIS-0613601 and CCF-0541036.

References

Magnus Carlsson and Thomas Hallgren. Fudgets: a graphical user interface in a lazy functional language. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 321–330, New York, NY, USA, 1993. ACM. ISBN 0-89791-595-X. doi: http://doi.acm.org/10.1145/165180.165228.

Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming, 4th International School, volume 2638 of LNCS*, pages 159–187. Springer-Verlag, 2003.

John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000. URL `http://www.cs.chalmers.se/~rjmh/Papers/arrows.ps`.

Monadic Programming Through Program Extraction

ICFP Poster Session Abstract

Josef Pohl

Computer Science Department, University of Wyoming, Laramie, WY

jpohl@cs.uwyo.edu

General Terms Program Extraction, Nuprl, Type Theory

Keywords Constructive Type Theory, Nuprl, Monad, Program Extraction

1. Purpose for Work

Closing the gap between formalized proof and practical programming has long been an objective of working in type theory. Creating mathematically correct software has been a long term goal of programming. There is a concerted effort build a strong relationship between theorem provers with a strong dependent type system and the utility of programming in a functional programming language such as Haskell, ML, and Ocaml amongst many others. There have been a number of languages such as Epigram (2), Cayenne (1), and Omega (5) that have approached the notion of program correctness by building a strong type system into a language. We approach this problem by using a theorem prover to formalize programming constructs.

Monads epitomize this effort and embody a powerful programming tool that can be used to incorporate side effects into purely functional programs. They are a formal construct but have revolutionized programming in purely functional environments. (The literature on this is much broader than the scope of this poster but we cite (3) and (6) as starting points although many others are available.) We play on this relationship by formalizing the structure of a monad as a type. By instantiating the type constructor and its associated operators in a dependent type theory we are able to reason about and program with monads as we would any other type. In a constructive context this reasoning narrows that gap between a formal proof and practical programming. The work of this presentation is done in Nuprl.

2. Nuprl

Nuprl is both a platform for theorem proving and programming and is a Martin-Löf type theory. As such Nuprl is a constructive type theory. It contains type universes and extensional function equality as well as a dependent type system. (The canonical reference for Nuprl is (4).) We use the related notions of propositions-as-types and proofs-as-programs to view a proof of a proposition as containing a program corresponding to the type of that proposition. A constructive proof of a proposition corresponds to a program much as a proposition itself corresponds to a type. These programs are terms in a purely functional programming language, namely the untyped λ-calculus with extensions. Computationally Nuprl's programming language is equivalent to more common functional programming languages such as Haskell.

3. Extracts

Extracting programs from proofs can be viewed as the process of deriving an implementation from a specification. We state what we want to do as a type or a proposition and synthesize how to do it, or the function or program that computes the answer. In the case of nuprl the synthesis takes place in the form of a proof that a given type is inhabited (or equivalently that a proposition has a proof.) In order to extract a program from a proof in Nuprl it must be first shown that a particular type is inhabited or equivalently that a proposition has a proof. Hence for some type T, we show $\vdash T$, a proof of which says that T is inhabited. In particular the type T is inhabited by a program P which has the type T. Nuprl's extraction mechanism can then be run on the resulting proof that T is inhabited giving a λ-term, which is the program P.

4. Monad Type

In order to program with monads in Nuprl we must first define the type of a monad. This process defines the structure of a monad, or the type, for which every instance of a monad, to be used as a monad, must conform. We define a monad as a type in Nuprl's type theory as a 6-tuple consisting of a type constructor M, the `>>=` and `return` operators, along with the three propositions stating the associativity and left and right identity laws for monads. The type constructor M is defined as a function type, $\mathbb{U} \rightarrow \mathbb{U}$. $\mathbb{U}$ is the designation for a universe of types at a particular level. In this way M is a polymorphic function from types to types. The operators `>>=` and `return` are similarly defined as function types. The `>>=` operator has the type, for arbitrary types A and B, $M\ A \rightarrow (A \rightarrow M\ B) \rightarrow M\ B$. `Return` has the type $A \rightarrow M\ A$. It is an obligation to assure, in Nuprl, that each component of the 6-tuple is in fact itself a well defined type.

The inclusion of the laws act as a constraint obligating the programmer to verify their validity. Hence in order to use instances of any monad a proof that the laws hold must be given. In the definition of the type we are stating these laws as predicates which are dependent upon the definitions of the type constructor, and the two operators, forming a dependent type. Given the correspondence between propositions (or predicates) and types it must be shown that the left and right hand side of the laws are equal in some type.

The well formedness proofs above do not contain any constructive content. In essence we are showing, in the case of the first three components of the monad type, that they inhabit a universe of types.

This is equivalent to saying that a type T is a member of $\mathbb{U}$, which is designated $T : \mathbb{U}$. The equalities are inhabited by the inhabitants of `Unit`, or `.`. Either they are equal under a given type or they are not. If one or both are not members of the type the proposition is absurd. The computational content, or a usable program or function, comes from the next step which is to show that `Monad` as a type is inhabited. Or in other words we must show $\vdash$ `Monad`

5. Instances

Much as one would define an instance of the monad class in a language such as Haskell by instantiating it we define an instance of a monad in Nuprl by showing that the type monad is inhabited. This is done exactly by giving a specific proof of $\vdash$ `Monad`. Each unique proof, with differing M, `>>=`, and `return`, gives a different instance of the `Monad` type, or in other words a different monad. Witnesses or well formedness proofs are provided for each of the first three components of the type. A witness in this case is the definition, or a specification, of the desired behavior. Separate proofs are then given that each component adheres to its respective laws. The resulting extract of the proof of the proposition `monad` is inhabited is a 6-tuple. It contains programs that are generated from each component in the original `monad` type. The programs for `>>=` and `return` are untyped λ-terms.

6. Examples

We provide several examples of this such as monads for Lists, State, and Maybe. In addition we give the constructs to use the individual portions of the instances. For instance the list monad is defined around the the type constructors for lists built into Nuprl. We create an abstraction with it to put it into the type of $\mathbb{U} \rightarrow \mathbb{U}$. The witnesses for the `>>=` and the `return` operator are the standard definition. The proof of each proceeds by unfolding the monad type and instantiating the component with the witness.

We then provide examples of programming with monads using our infrastructure. We introduce a do-notation and give a number of real, albeit, simple examples of extracting monad based programs using the instances we have provided. By using an instance of a monad to prove a specification, or a proposition (type) stating the properties, of a program we can generate an extract, which has the monadic properties and behavior embedded in it.

7. Ongoing Work

As this is ongoing work, we describe our continuing efforts to make this an efficient and useful technique for programming in a constructive type theory environment as well as providing utilities that build programs for use in a more traditional setting. In this vein we are building a translator from Nuprl's term language into the syntax of Haskell. Examples of this work will be shown. This is done at a meta-level in the built-in language (a variant of ML) used to define the rules and tactics in Nuprl. It is a strict translation of the lambda terms into the syntax of Haskell. Further we will explain our ongoing work in generating monad based tactics that facilitate writing programs, a la induction techniques.

References

[1] Lennart Augustsson Cayenne - a Language with Dependent Types in *International Conference on Functional Programming*,pages 239–250. 1998

[2] Conor McBride and James McKinna The view from the left. in *Journal of Functional Programming*, pages 69 – 111, 14, 1, Cambridge Univ. Press, 2004.

[3] Eugenio Moggi Computational Lambda-Calculus and Monads in *Proceedings 4th Annual IEEE Symp. on Logic in Computer Science, LICS'89.* Pages 14 – 23, IEEE, 1989.

[4] Constable, R. L. and S. F. Allen and H. M. Bromley and W. R. Cleaveland and J. F. Cremer and R. W. Harper and D. J. Howe and T. B. Knoblock and N. P. Mendler and P. Panangaden and J. T. Sasaki and S. F. Smith Implementing Mathematics with the Nuprl Development System. Prentice-Hall,1986

[5] Tim Sheard Putting Curry-Howard to Work. in *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 74–85, ACM, 2005.

[6] Pillip Wadler Monads for functional programming in *POPL '92 :Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1 – 14, ACM 2002

Unified Type Checking for Type Classes and Type Families

Tom Schrijvers *
K.U.Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Martin Sulzmann
ITU, Denmark
martin.sulzmann@gmail.com

Keywords Haskell, type checking, type classes, type families

1. Introduction

Haskell has a rich type system with various complementary, interacting and overlapping features. In particular we think of the established *type classes*, with several extensions: multiple parameters, functional dependencies (Jones 2000), ... In a recent proposal (Schrijvers et al. 2008), a new feature is added to the Haskell language: type-level functions, or *type families* in GHC.

This multitude of type-level features is a blessing for programmers. A well-chosen combination of features allows the accurate expression of many problem domain semantics.

However, the plethora of features is also a nightmare for Haskell compiler writers, who have to implement and maintain all these features. For instance, GHC's core type checking modules for type classes and type families comprise approximately 3.1 kLoC and 1.2 kLoC, which are understood by very few people. Currently, no other Haskell system has managed to provide the same type class functionality as GHC.

The contributions of this work aim at reducing the implementation complexity of two Haskell type system features, type classes and type families:

- We reduce the type checking problem of type classes to a type checking problem of type functions (Section 2).
  - We propose a small extension of the current type checking algorithm for type functions to cope with the above mapping.
- Thus, we can lift two current restrictions of type classes (Section 3).
  - Type functions and type classes can be freely mixed in the instance context.
  - Instance declarations can be applied both ways. That is, dictionaries of the instance contexts can be extracted from the instance head dictionary.

In the following, we illustrate the above points.

* Post-doctoral researcher of the Fund for Scientific Research - Flanders.

2. Unified Type Checking

The core of our idea is simple Functional Programming wisdom[1]:

> A predicate is (nothing more than) a function that returns a boolean value.

Hence, our idea is to represent a type class (a type-level predicate) as a type family (type-level function) returning a *type-level boolean*. These type-level booleans are represented by empty types:

```
data TRUE
data FALSE
```

Consider for instance, the `Ord` type class with some instances (methods omitted):

```
class     Eq a          => Ord a
instance                   Ord ()
instance (Ord a, Ord b) => Ord (a,b)
```

These are mapped onto the following type families:

```
type family   OrdF a
type instance OrdF a = AND (EqF a) (OrdI a) -- (*)

type family   OrdI a
type instance OrdI ()    = TRUE
type instance OrdI (a,b) = AND (OrdF a) (OrdF b)
```

The encoding `OrdF a` reduces to `TRUE` iff `a` is instance of the `Ord` type class. If it is not, then `OrdF a` does not reduce to a value. The generic case (*) states that the class context must be satisfied and there must be an instance. The family `OrdI` captures the instances. These instances may recursively call `OrdF` to satisfy their contexts.

In order to verify whether a type class constraint $C\ \bar{t}$ holds, we verify whether its family encoding $CF\ t$ reduces to `TRUE`. In other words, we verify the equality constraint $CF\ t$ ~ `TRUE`. For this purpose, we can simply use the rewriting-based algorithm of type families (Schrijvers et al. 2008). It only needs one more equality rewriting rule to decompose the new `AND` symbol:

$$\texttt{AND}\ t_1\ t_2 \sim \texttt{TRUE} \quad \rightarrowtail \quad t_1 \sim \texttt{TRUE},\ t_2 \sim \texttt{TRUE}$$

Hence, we can make the following reduction for `Ord ((),())`:

```
OrdF ((),()) ~ TRUE
>-> AND (EqF ((),())) (OrdI ((),())) ~ TRUE
>-> EqF ((),()) ~ TRUE, OrdI ((),()) ~ TRUE
>->* AND (OrdF ()) (OrdF ()) ~ TRUE
>-> OrdF () ~ TRUE, OrdF () ~ TRUE
>->* TRUE ~ TRUE, TRUE ~ TRUE
```

[1] Often countered by the opposite Logic Programming wisdom.

3. Additional Expressiveness

3.1 Calling Type Functions in Instance Contexts

A common pattern in type-level programming is to select an instance based on a previous type-level computation. Here, we consider comparison among type-level natural numbers.

```
data Zero   = Zero
data Succ n = Succ n

type family EqTest
type instance EqTest Zero Zero = TRUE
type instance EqTest (Succ x) (Succ y) = EqTest x y
type instance EqTest Zero (Succ x) = FALSE
type instance EqTest (Succ x) Zero = FALSE
```

Suppose we have a binary type class `C` whose actual implementation is "selected" based on a (type-level) test among its argument.

```
instance Selector x y (EqTest x y) => C x y
```

In the above, we assume the helper class `Selector` which will select the appropriate instance. At the moment, GHC cannot properly deal with such cases because of "incompatible" checking mechanisms used for type classes and type functions. None of these problems arises anymore in the unified type checking approach.

3.2 Bi-directional Instance Declarations

The Current Situation The most widely used implementation approach for type classes is the dictionary-passing translation (Hall et al. 1996). This approach only allows instance declarations to be applied in a single direction. That is, we can build the dictionary for the instance head given the dictionaries for the instance context, but not the other way around. For example, in case of

```
instance (Eq a, Eq b) => Eq (a,b)
```

we can build the `Eq (a,b)` dictionary given dictionaries for `Eq a` and `Eq b` but we cannot extract `Eq a` and `Eq b` from `Eq (a,b)`. Is this a serious restriction?

For example, this program cannot be dealt with:

```
cmp :: Eq (a,b) => a -> a -> b -> b -> Bool
cmp x1 x2 y1 y2 = x1 == x2  && y1 == y2
```

The program text gives rise to `Eq a` and `Eq b`, but we cannot construct the necessary dictionaries given `Eq (a,b)`. Aware of the limitation of the dictionary passing translation, current type checkers reject this program.

Of course, this is a artificial example because we expect that the programmer provides the more sensible signature

```
cmp :: (Eq a,Eq b) => a -> a -> b -> b -> Bool
```

New Problems While in the past, with a relatively simple type system, the dictionary-passing limitations could we lived with. This is no longer the case. In the presence of GADTs it can happen that a sensible type annotation provided by the programmer is refined to the problematic case.

Consider

```
data T a where
   Pair :: b -> c -> T (b,c)

instance Eq a => Eq (T a) where
   (==) (Pair x1 x2) (Pair y1 y2) =
                    x1 == x2  && y1 == y2
```

The instance context provides `Eq a` which is refined to `Eq (b,c)` because of the GADT pattern match. However, the program text demands `Eq b` and `Eq c`.

It is possible to lift the restriction in our unified approach. As type function equations are symmetric, type class checking naturally works both ways (bi-directionally) for instance declarations.

A Simple Solution To properly support bi-directional instance declarations we need to make one adjustment to the existing dictionary-passing translation scheme. Each dictionary carries in addition to method definitions a context which refers to the possibly dicionaries in the instance context.

```
type family ContEq a

data DictEq a = E { (==)    :: a -> a -> Bool
                  , context :: ContEq a }
```

The concrete type of `context` depends on the specific instance head and can easily provided by appropriate type function definitions for each instance declaration. For our running example

```
instance (Eq a, Eq b) => Eq (a,b)
```

we generate

```
type instance ContEq (a,b) = (DictEq a,DictEq b)
```

4. Conclusion & Future Work

We have sketched a simplified approach to dealing with both type classes and type families with the same unified type checking algorithm. The unified approach yields a significantly more expressive system. In future work we intend to elaborate – in particular with respect to evidence handling – and implement the approach, and prove it correct.

References

C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.

Mark P. Jones. Type classes with functional dependencies. In *Proc. of ESOP 2000*, number 1782 in Lecture Notes in Computer Science. Springer-Verlag, 2000.

Tom Schrijvers, Simon Peyton-Jones, Manuel Chakravarty, and Martin Sulzmann. Type Checking with Open Type Functions. In *International Conference on Functional Programming*, 2008. Accepted.

Normalization in the Dual Calculus with Sigma Reductions

Jeffrey A. Vaughan Stephanie Weirich Steve Zdancewic
University of Pennsylvania
vaughan2@seas.upenn.edu {sweirch, stevez}@cis.upenn.edu

Introduction

Dual calculus (Wadler 2003, 2005) is a programming typed language whose expression syntax consists of three sorts: *terms*, *coterms*, and *statements*. Terms—which include constructors and variables—intuitively correspond to data; coterms—which include elimination forms and *covariables*—correspond to evaluation contexts. Terms have normal-forms called *values* and coterms have normal forms called *covalues*. Statements represent computations. They are composed of a term m and a coterm k, and written $m \bullet k$.

Special expressions allow variable and covariable abstraction over statements. The latter, $(m \bullet k).\alpha$, is a term corresponding to the familiar let/cc operator. The former, $x.(m \bullet k)$, is precisely dual and is called (inspired by Lovas and Crary (2006)) let/ct—"let-with-current-term." Let/cc abstractions are not values, nor are let/ct abstractions covalues.

Statements can reduce in two ways. β reductions may occur in a statement when the term's top-level constructor fits the coterm's elimination form. For instance, the rule

$$\langle m, n\rangle \bullet \mathsf{fst}[k] \rightarrow_{\beta\wedge} m \bullet k$$

states that a projection (coterm) containing sub-coterm k in juxtaposition with a pair (term) made up of m and n steps to a new statement: $m \bullet k$. Statements may also step using ς rules. These lift non-normal components of terms (dually coterms) closer to a statement's top level. For example, using ς (and reducing a resulting administrative β-redex) gives the following:

$$\langle m, n\rangle \bullet k \rightarrow^* m \bullet x.(\langle x, n\rangle \bullet k)$$

Intuitively this reduction sequence is starting to evaluate the $\langle m, n\rangle$ pair by building a stack frame ready to accept the result of computing m. Earlier work has also considered η-expansion, but we do not do so here.

Some statements may reduce by both ς and β rules. Wadler (2003) demonstrates that particular dual calculus evaluation strategies correspond to call-by-value or call-by-name evaluation in lambda calculus. Consider statement $m \bullet k$; these strategies can be summarized as follows.

call-by-value: If m is a value or let/cc, reduce by β. Otherwise, simplify m using ς-reductions.

call-by-name: If k is a covalue or let/ct, reduce by β. Otherwise, simplify k using ς-reductions.

In call-by-value dual calculus, normal-form statements always have form $v \bullet k$, where v is a value. Dually, call-by-name normal-form statements have from $m \bullet p$ where p is a co-value. These properties are desirable when dual calculus is used to model lambda calculus reduction (Wadler 2003) or lambda-mu calculus equivalences (Wadler 2005).

Wadler conjectured that dual calculus with only β rules is strongly normalizing. Dougherty et al. (2005) proved strong normalization for β-reduction with additional structural rules. Lovas and Crary (2006) formalized a proof proof of weak normalization for a similar system in the TWELF logical framework. Tzevelekos (2006) gives a reducibility candidates argument for strong normalization of dual calculus with β, η and ς reductions.

This work presents a logical relations proof of strong normalization for for call-by-value (and dually call-by-name) dual calculus with β and ς reductions. We also examine a *mixed-reduction* strategy in which each statement carries a flag indicating whether to evaluate it in a call-by-name or call-by-value fashion. We show the existence of non-terminating statements under mixed-reduction.

Dual calculus expressions may be interpreted as proofs in LK, a classical sequent calculus. (This is reminiscent of the Curry-Howard correspondence relating lambda calculus terms and natural deduction proofs.) Type assignment for terms corresponds to sequent calculus right rules, coterms correspond to left rules, and statements correspond to the cut rule.

It is trivial to show choice of reduction strategy does not effect which sequents are provable. This leads to the surprising observation that infinite-reduction sequences are not "logically harmful"—that is, do not cause logical inconsistencies—in dual calculus. This result joins others (e.g. Urban (2000)) showing that the relation between terms, types, and normalization for classical sequent systems is more subtle than for natural deduction systems studied with lambda calculus.

Dual Calculus is Strongly Normalizing

This section will sketch a logical-relations proof of strong normalization for dual calculus. Defining the logical relation and proving the main lemma will require several auxiliary definitions: language syntax, static and dynamic semantics, the logical relation itself, and a auxiliary notion of *logical substitution*.

The following grammar defines the syntax of dual calculus.

Types	A, B	$::=$	$X \mid A \wedge A \mid A \vee A \mid \neg A$
Terms	m, n	$::=$	$x \mid \langle m, n\rangle \mid [k]\mathsf{not} \mid \langle m\rangle\mathsf{inl} \mid \langle n\rangle\mathsf{inr} \mid (S).\alpha$
Coterms	k, l	$::=$	$\alpha \mid \mathsf{fst}[k] \mid \mathsf{snd}[l] \mid \mathsf{not}\langle m\rangle \mid [k, l] \mid x.(S)$
Statements	S	$::=$	$m \bullet k$

Some terms are also values. Intuitively values are terms in which all let/ct subterms are "suspended" by an enclosing not. Formally the set of call-by-value values is generated by the following grammar.

$$\text{Values} \quad v, w \quad ::= x \mid \langle v, w\rangle \mid [k]\mathsf{not} \mid \langle v\rangle\mathsf{inl} \mid \langle w\rangle\mathsf{inr}$$

The dynamic semantics for call-by-value dual calculus uses term evaluation contexts, written E. Term evaluation contexts are defined by

$$E ::= \langle\{\}, n\rangle \mid \langle v, \{\}\rangle \mid \langle\{\}\rangle\mathsf{inl} \mid \langle\{\}\rangle\mathsf{inr}$$

where n is not be a value. The symbol $\{\}$ represents a hole which may be filled by a term. $E\{m\}$ denotes the term created by replacing E's hole with m.

The call-by-value dynamic semantics are as follows.

$$\langle v, w\rangle \bullet \mathsf{fst}[k] \to_\beta v \bullet k \qquad \langle v, w\rangle \bullet \mathsf{snd}[l] \to_\beta w \bullet l$$

$$\langle v\rangle\mathsf{inl} \bullet [k, l] \to_\beta v \bullet k \qquad \langle w\rangle\mathsf{inr} \bullet [k, l] \to_\beta w \bullet l$$

$$[k]\mathsf{not} \bullet \mathsf{not}\langle m\rangle \to_\beta m \bullet k$$

$$v \bullet x.(S) \to_\beta \{v/x\}S \qquad (S).\alpha \bullet k \to_\beta \{k/\alpha\}S$$

$$E\{m\} \bullet k \to_\varsigma (m \bullet x.(E\{x\} \bullet \beta)).\beta \bullet k$$

We use a logical-relations argument to show that call-by-value dual calculus is strongly normalizing. The following four indexed sets represent unary relations on dual calculus expressions. The main lemma will demonstrate that all well-typed expressions are logical—that is, are members of a relation. Strong normalization follows as a corollary.

$$S \in S[\![\#]\!] \quad \text{iff} \quad S \to^* S' \not\to$$

$$\begin{array}{rll} x \in V[\![A]\!] & & \text{(always)} \\ \langle v, w\rangle \in V[\![A \wedge B]\!] & \text{iff} & v \in V[\![A]\!] \text{ and } w \in V[\![B]\!] \\ \langle v\rangle\mathsf{inl} \in V[\![A \vee B]\!] & \text{iff} & v \in V[\![A]\!] \\ \langle w\rangle\mathsf{inr} \in V[\![A \vee B]\!] & \text{iff} & w \in V[\![B]\!] \\ [k]\mathsf{not} \in V[\![\neg A]\!] & \text{iff} & k \in K[\![A]\!] \end{array}$$

$$k \in K[\![A]\!] \quad \text{iff} \quad \text{for all } v \in V[\![A]\!], v \bullet k \in S[\![\#]\!]$$

$$m \in M[\![A]\!] \quad \text{iff} \quad \text{for all } k \in K[\![A]\!], m \bullet k \in S[\![\#]\!]$$

We can make several observations about this family of relations. First, they are defined on open terms, without closing substitutions. This approach is useful because dual calculus, like LK, has no closed well-typed terms. Second, $V[\![A]\!]$ and $K[\![A]\!]$ are defined by mutual recursion on their type parameters, however the "recursive call" in $K[\![A]\!]$ does not mention a structurally smaller type. Instead, the definition of $V[\![A]\!]$ must be unfolded to verify that the recursion is well-founded. Third, it is while type indexed, the relation family is otherwise oblivious to types; it does not mention a context, and contains expressions which cannot be typed in any context. This is useful as the main lemma does not depend on subject reduction (a property dual calculus enjoys, but for which we know no published proof). Fourth, $M[\![\cdot]\!]$ is defined by universally quantifying over $K[\![\cdot]\!]$, and $K[\![\cdot]\!]$ by universally quantifying over $V[\![\cdot]\!]$. This strikingly similar to Pitts's (2005) TT-closure method for reasoning about program equivalence.

Before stating the main lemma, we need two more definitions.

Space constraints prevent us from giving the static semantics for dual calculus. We follow the presentation of Wadler (2005), which simplifies the original system (Wadler 2003) by liberalizing the cut and identity rules and by removing the now superfluous structural rules. Dual calculus is typed using three mutually inductive relations. Each includes an *antecedent* context Γ and a *succedent* context Θ. Antecedent contexts map variables to types, and succedent contexts map covariables to types. For call-by-value we can read the typing judgments as follows.

- Right Sequent, $\Gamma \vdash m : A; \Theta$. Term m has type A in contexts Γ and Θ. Intuitively, evaluating m might yield a return value at type A or it might throw to a continuation described by Θ.
- Left Sequent, $\Gamma; k : A \vdash \Theta$. Coterm k is a continuation which can accept a value of type A.
- Center Sequent, $S : (\Gamma \vdash \Theta)$. Statement S is a computation which takes one value per binding in Γ as input. S yields only a single output, whose type is included in Θ.

Substitutions are partial maps that take variables to terms and covariables to coterms. Some substitutions have the special property of replacing (co)variables with logical (co)terms. We say substitution σ is Γ, Θ-logical when

(i) for all $x \in \mathrm{dom}(\sigma)$, $\sigma(x) \in V[\![\Gamma(x)]\!]$, and

(ii) for all $\alpha \in \mathrm{dom}(\sigma)$, $\sigma(\alpha) \in K[\![\Theta(\alpha)]\!]$.

Lemma (Main Lemma)**.** *Suppose $\mathcal{D}$ is a typing derivation, then*

- *if $\mathcal{D} :: S : (\Gamma \vdash \Theta)$ then for all Γ, Θ-logical σ, $\sigma(S) \in S[\![\#]\!]$,*
- *if $\mathcal{D} :: \Gamma \vdash v : A; \Theta$ then for all Γ, Θ-logical σ, $\sigma(v) \in V[\![A]\!]$,*
- *if $\mathcal{D} :: \Gamma \vdash m : A; \Theta$ then for all Γ, Θ-logical σ, $\sigma(m) \in M[\![A]\!]$,*
- *if $\mathcal{D} :: \Gamma; k : A \vdash \Theta$ then for all Γ, Θ-logical σ, $\sigma(k) \in K[\![A]\!]$.*

Proof Sketch. By induction on the structure on the $\mathcal{D}$. Note the lemma deeply nests the quantification on σ. This is required to get a strong induction hypothesis for the $\mathcal{D} :: \Gamma \vdash (S).\alpha : A; \Theta$ case. Many cases use the following fact: if $S \to S'$ and $S' \in S[\![\#]\!]$ then $S \in S[\![\#]\!]$. The full proof also requires one-off lemmas showing that various sigma reductions involving elements of $M[\![\cdot]\!]$, $V[\![\cdot]\!]$ and $K[\![\cdot]\!]$ produce elements of $K[\![\cdot]\!]$. □

Corollary (Strong normalization)**.** *Every well-typed dual calculus statement is strongly normalizing.*

Proof. Suppose $S : (\Gamma \vdash \Theta)$, then $S \in S[\![\#]\!]$ by the main lemma. Hence $S \to^* S' \not\to$. □

Mixed Reduction is Not Strongly Normalizing

Mixed reduction is a natural variant of dual calculus. In mixed reduction every statement carries a tag indicating whether its top-level redex should reduce using call-by-name or call-by-value rules. For instance $m \overset{\leftarrow}{\bullet} k \to_{mixed} S$ when $m \bullet k \to_\beta S$ in call-by-value. (And similarly $m \overset{\rightarrow}{\bullet} k$ reduces when there a suitable call-by-name reduction.) The ς-reductions include

$$E\{m\} \overset{\leftarrow}{\bullet} k \to (m \overset{\leftarrow}{\bullet} x.(E\{x\} \overset{\leftarrow}{\bullet} \beta)).\beta \overset{\leftarrow}{\bullet} k$$

and its dual.

While call-by-value (dually call-by-name) dual calculus is both strongly normalizing and deterministic, mixed reduction is not strongly normalizing. For a counterexample, the statement $z \overset{\rightarrow}{\bullet} [\alpha, x.(\langle y, x\rangle \overset{\leftarrow}{\bullet} \beta)]$ reduces to itself in six steps.

References

Daniel J. Dougherty, Silvia Ghilezan, Pierre Lescanne, and Silvia Likavec. Strong normalization of the dual classical sequent calculus. In *LPAR*, pages 169–183, 2005.

William Lovas and Karl Crary. Structural normalization for classical natural deduction. Available from `http://www.cs.cmu.edu/~wlovas/papers/clnorm.pdf`, 2006.

A. M. Pitts. Typed operational reasoning. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 7, pages 245–289. The MIT Press, 2005. ISBN 0-262-16228-8.

Nikos Tzevelekos. Investigations on the dual calculus. *Theor. Comput. Sci.*, 360(1):289–326, 2006. ISSN 0304-3975.

Christian Urban. *Classical Logic and Computation*. PhD thesis, University of Cambridge, October 2000.

Philip Wadler. Call-by-value is dual to call-by-name. In *ICFP '03*, Uppsala, Sweden, 2003.

Philip Wadler. Call-by-value is dual to call-by-name, reloaded. In *Rewriting Techniques and Applications '05*, Nara, 2005.

Translucent Abstraction

Safe Views through Bidirectional Transformation

Meng Wang and Jeremy Gibbons

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
{menw,jg}@comlab.ox.ac.uk

The *view-update* problem found in database research has been a topic of study for decades. It is essentially about mapping updates on *views* of data back to the physical data in a legitimate manner. As an example of the view-update problem, we might have some tree-structured source data, which we project into a linear abstract value through a *get* function. Now we might want to update the abstract value; the challenge then is to come up with a backward function *put*, as the inverse of *get*, to put the updates in the abstract value back into the source value.

The keys to solutions to the view-update problem are *bidirectional transformation techniques*, which allow computations to be inverted. Many approaches have been proposed. One is to provide a fixed set of invertibility-preserving combinators for transformations. However, this combinator approach is quite limited in expressiveness, and not very intuitive to use. For more general applications, we could try to derive a backwards version of a transformation that is written in a more general language. Very often, a *get* function from a source to an abstract value

$$get :: S \to V$$

is an abstraction that loses information. It is generally impossible to reconstruct the source with an inverse function

$$get^{-1} :: V \to S$$

Thus, almost all bidirectional systems try instead to come up with a backward transformation of the form

$$put :: (V, C) \to S$$

where C (known as a *complement*) supplies information that is lost during the transformation and is used to rebuild the source.

Independently, a concept of *views* has developed in the area of programming language design (for example, by Wadler). A view is an abstraction of data's actual implementation, which provides a programming interface that is more convenient to use and more robust to changes. With a pair of conversion functions, data can be converted *to* and *from* a view. Views provide a powerful mechanism, which reconciles abstraction and pattern matching. On one hand, views are decoupled from the actual implementation of data; on the other hand, pattern matching and equational reasoning are readily supported. We term this kind of abstraction *translucent*, in contrast to the transparent nature of algebraic datatypes and the opaque nature of abstract datatypes.

Conventionally, view implementers are required to come up with a pair of conversion functions that are each other's inverses, a condition that is difficult to check and maintain. This non-machine-checkable condition is a serious weakness of the original design of views, which was considered impractical and was never implemented. 'Safe' variants of views have been proposed by many authors since; this is still an active research area. To circumvent the problem of equational reasoning, most approaches restrict the use of view constructors to patterns, and do not allow them to appear on the right-hand side of an equation.

We argue that the problem of views in programming languages is closely related to the view-update problem: solutions of one problem can be used for the other. Traditionally, research into the view-update problem focuses on preserving data and its associations rather than on structure. This is because most database data is stored in simple and loosely specified structures, the preservation of which is either simple or less important. However, the situation is different in functional languages, where we frequently make use of rich datatypes and write functions that transform between them. As a result, we propose a bidirectional system that focuses on structure preservation.

Specifically, we:

- extend previous bidirectional transformation frameworks by accepting a wider range of programmer-defined *get* functions on general datatypes; a *put* function is automatically generated, so as to be well-typed and law-abiding;
- enlarge the domain of the backward transformation to deal with arbitrary updates on abstract values that fall within the range of the *get* function, through extending the framework with a function *create*, in addition to *put*;
- make use of our bidirectional transformation technique in the field of programming language design, specifically views of datatypes, resulting in a system that is convenient (programmers only need to specify a one-directional transformation), correct (correctness of equational reasoning on views can be validated), and safe (poorly designed views are detected at the earliest stage).

Bidirectional Transformation

The basic idea of our bidirectionalization technique is the *constant complement approach*. A function $cpt :: S \to C$ is a complement function to a *get* function $get :: S \to V$ iff the tupled function $get \vartriangle cpt :: S \to (V, C)$ is injective.

Intuitively, the complement function keeps whatever information was lost during the *get* process in a complement value, so that when this is tupled with the abstract value, the two uniquely determine the source value. As a simple example, for the *get* function $fst :: (a, b) \to a$, the function $snd :: (a, b) \to b$ is a complement function, remembering the second element of the input pair. Another example is a *get* function that flattens trees into lists; the complement function will need to preserve the exact shape of the tree, but may discard leaf values.

The complement functions are usually not unique; for example, in addition to *snd*, both $id :: a \to a$ and $swap :: (a, b) \to (b, a)$ are

complement functions of *fst*. There are multiple ways of encoding the shape of a tree as well.

After obtaining the injective tupling of the *get* and complement functions, reversing this for a backward function becomes straightforward. We illustrate the complete process through an example. As a notational convention, we superscript *get* functions by '$<$', complement functions by '$\bullet$', tupled functions by '$\lessdot$', *put* functions by '$>$', and *create* functions by '$\succ$'.

Consider the *get* function $append^<$ defined as:

$$\textbf{data}\ List\ a = Nil \mid Cons\ a\ (List\ a)\ \textbf{deriving}\ Show$$

$$\begin{aligned} append^<\ (Nil, ys) &= ys \\ append^<\ (Cons\ x\ xs, ys) &= Cons\ x\ xys \\ \quad \textbf{where}\ xys &= append^<\ (xs, ys) \end{aligned}$$

A complement function is constructed by memoizing the computation, in this case, the deconstruction of the first input list.

$$\begin{aligned} append^\bullet\ (Nil, ys) &= C_1 \\ append^\bullet\ (Cons\ x\ xs, ys) &= C_2\ (append^\bullet\ (xs, ys)) \\ \quad \textbf{where}\ xys &= append^<\ (xs, ys) \end{aligned}$$

(The redundant **where** clause is an artifact of our translation.) A new complement type C with constructors C_1 and C_2 is introduced:

$$\textbf{data}\ C = C_1 \mid C_2\ C\ \textbf{deriving}\ Show$$

Basically, a C value encodes the length of a list; for example, $C_2\ (C_2\ (C_2\ C_1))$ represents a list of length 3. When tupled with the *get* function, we obtain an injective function $append^\lessdot$: the result of the concatenation and the length of the first list uniquely determines the input pair.

$$\begin{aligned} append^\lessdot\ (Nil, ys) &= (ys, C_1) \\ append^\lessdot\ (Cons\ x\ xs, ys) &= (Cons\ x\ xys, C_2\ c) \\ \quad \textbf{where}\ (xys, c) &= append^\lessdot\ (xs, ys) \end{aligned}$$

Note that complement function calls on the right-hand side are subsumed by tupled function calls in the **where** clause. Now, the task of creating a backward function reduces to swapping patterns between the two sides of the equations.

$$\begin{aligned} append^>\ (ys, C_1) &= (Nil, ys) \\ append^>\ (Cons\ x\ xys, C_2\ c) &= (Cons\ x\ xs, ys) \\ \quad \textbf{where}\ (xs, ys) &= append^>\ (xys, c) \end{aligned}$$

Here, $append^>$ is the *put* function generated for $append^<$. When given an abstract value, in this case a list, and a corresponding complement, $append^>$ splits the list at the position where the two source lists were joined.

Implementing Views

The bidirectional system described above can be used to derive one-directional conversion functions from their programmer-specified counterpart. Let's consider a simple example of implementing queue structures. With algebraic datatypes, we could define the following, which is isomorphic to the list datatype.

$$\textbf{data}\ Queue\ a = E \mid Q\ a\ (Queue\ a)$$

Suppose that later we want to alter the implementation by always separating out the first two elements in the queue. A more appropriate design will be declaring *Queue* as a view, so that it becomes more resilient to implementation changes.

$$\begin{aligned} &\textbf{view}\ (List\ a, List\ a) = E \mid Q\ a\ (List\ a, List\ a) \\ &\quad to = appendQ^< \end{aligned}$$

We follow Wadler's view syntax by not giving explicit names to views. (This choice is orthogonal to our proposal.) The constructors E and Q become an interface to the queue structure, whose underlying implementation is a pair of lists. A one-directional conversion function *to* is specified as a slight variant of the $append^<$ function above. Our bi-directional system automatically generates the backward function *from*. We can now program to the interface and be oblivious to the implementation. For example, we can write

$$\begin{aligned} insert\ x\ E &= Q\ x\ E \\ insert\ x\ (y : ys) &= Q\ y\ (insert\ x\ ys) \end{aligned}$$

and apply it to queue values

$$x = insert\ 3\ (Q\ 1\ (Q\ 2\ E))$$

Views can then be translated into primitive language constructs.

$$\begin{aligned} &\textbf{data}\ Queue\ a = E \mid Q\ a\ (Queue\ a) \\ &to :: (List\ a, List\ a) \rightarrow Queue\ a \\ &to = appendQ^< \\ &from :: (Queue\ a, C) \rightarrow (List\ a, List\ a) \\ &from = appendQ^> \\ &insertv\ x\ E = Q\ x\ E \\ &insertv\ x\ (y : ys) = Q\ y\ (insertv\ x\ ys) \\ &insert\ q = from\ (insertv\ (to\ q), appendQ^\bullet\ q) \\ &x = insert\ 3\ (from\ (Q\ 1\ (Q\ 2\ E), C_2\ (C_2\ C_1))) \end{aligned}$$

A view declaration is turned into a datatype and given a fresh name. The function *insert* on the view is translated into a function *insertv* that operates on the view datatype *Queue a*. The *from* function is the generated backward transformation of *to*. We have inserted type annotations for the purposes of illustration. Function *insert* now takes in a pair of lists, converts it to a queue, inserts an element, and finally converts the extended queue back to a pair of lists. Note that we manually inserted the complement value $C_2\ (C_2\ C_1)$ when the term $Q\ 1\ (Q\ 2\ E)$ is first constructed, to enforce the property that only two elements are separated out in the first list of the pair. In more general cases, this complement value does not exist. We refer the reader to our draft paper for techniques to handle partial or incorrect complements.

As an example, the evaluation in source syntax of the expression $insert\ 3\ (Q\ 1\ (Q\ 2\ E))$ in view syntax proceeds as follows. (We deliberately keep all the conversions explicit for demonstration purposes. Program fusion can help to eliminate unnecessary conversions.)

$$\begin{aligned} x &= insert\ 3\ (from\ (Q\ 1\ (Q\ 2\ E), C_2\ (C_2\ C_1))) \\ &\ldots \\ &= insert\ 3\ ((Cons\ 1\ (Cons\ 2\ Nil)), Nil) \\ &= from\ (insertv\ 3\ (to\ (Cons\ 1\ (Cons\ 2\ Nil)), Nil)), \\ &\qquad appendQ^\bullet\ ((Cons\ 1\ (Cons\ 2\ Nil)), Nil)) \\ &\ldots \\ &= from\ (insertv\ 3\ (Q\ 1\ (Q\ 2\ E)), C_2\ (C_2\ C_1)) \\ &\ldots \\ &= from\ (Q\ 1\ (Q\ 2\ (Q\ 3\ E)), C_2\ (C_2\ C_1)) \\ &= (Cons\ 1\ (Cons\ 2\ Nil), (Cons\ 3\ Nil)) \end{aligned}$$

Applying the view to $(Cons\ 1\ (Cons\ 2\ Nil), (Cons\ 3\ Nil))$ gives $Q\ 1\ (Q\ 2\ (Q\ 3\ E))$, which is precisely the intended result of inserting 3 into $Q\ 1\ (Q\ 2\ E)$. Instead of going in great detail into the conversions between view and source datatypes as to the derivation above, we could reason on the view level as if the views were datatypes.

$$\begin{aligned} &insert\ 3\ (Q\ 1\ (Q\ 2\ E)) = Q\ 1\ (insert\ 3\ (Q\ 2\ E)) \\ &\quad = Q\ 1\ (Q\ 2\ (insert\ 3\ E)) = Q\ 1\ (Q\ 2\ (Q\ 3\ E)) \end{aligned}$$

The correctness of reasoning on the view level follows from bidirectional laws, which are observed by our bidirectional transformation system. More details of the result can be found in a draft paper at `http://web.comlab.ox.ac.uk/people/Meng.Wang/`.

Schema Mapping Polymorphism

Ryan Wisnesky
Harvard University
ryan@cs.harvard.edu

This poster presents type-theoretic, functional-programming inspired enhancements to the theory and practice of schema mapping.

Overview

Schema mappings are logical expressions in carefully crafted formalisms that express invariants between data represented in different schemas (Popa et al. 2002). These expressions are often created automatically by a "mapping generator" that uses as input source and target schemas and a set of "correspondences" between source and target schema elements (Miller et al. 2000; Melnik et al. 2005; Bonifati et al. 2005). Figure 1 shows IBM's Clio mapping tool (Haas et al. 2005) in action.

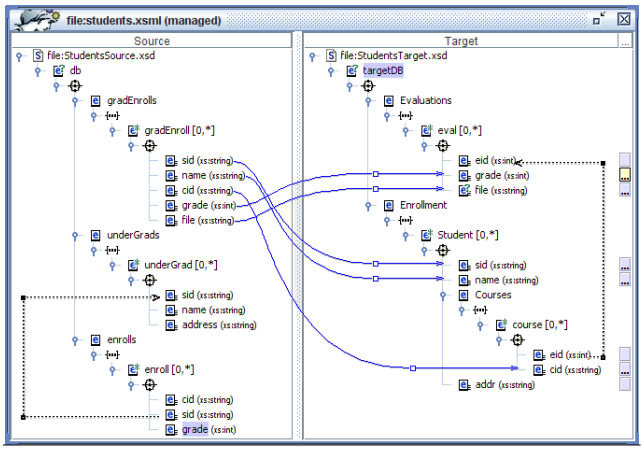


Figure 1. A schema mapping in Clio

In this screenshot, the user has loaded a source and a target schema and has entered a number of correspondences between atomic-level elements of both schemas. Clio generates a set of *schema mapping expressions* from this simple input of schemas and correspondences. The generated (schema) mapping expressions can then be converted into a semantics-preserving query that transforms data from the source schema to the target schema. Our notion of schema can describe both (non-recursive) XML and relational data, and queries can be generated in a number of target languages (SQL, XSLT, etc). When query generation is viewed as compilation, mapping languages correspond to intermediate forms.

Mapping tools are typically used when semantics-preserving data transformation is needed but users cannot or do not want to write queries themselves (Miller et al. 2000); for instance, in a business context where non-programmers need to migrate information between departmental databases. Moreover, it can be difficult to manually create semantics preserving queries in the presence of schema integrity constraints (e.g. foreign keys). Models and semantics of schema mappings for data exchange (Fagin et al. 2003) and operations over mappings (Melnik et al. 2003; Fagin et al. 2005, 2007) have been extensively studied by the database community.

Motivation

Current mapping systems suffer from a number of mapping-reuse related drawbacks that are similar to challenges previously encountered by the functional programming community:

- Reliance on concrete schemas, so users that have created mapping expressions from S to T cannot re-use them at related schemas S', T' even when that is a theoretically valid use. This is the *mapping polymorphism* problem.
- Mapping formalisms cannot express mappings that depend on other mappings. For instance, a user may map from S to T and then copy the mapping and add an extra correspondence; if the original mapping changes, this change is not propagated to the new mapping. This is the *mapping dependence* problem.
- Given a set of mapping expressions M and schemas S and T, mapping tools cannot (in general) construct a mapping from S to T. Moreover, there is no way to determine if M can be used with *any* schemas. This is the *mapping inference* problem.

The problems are particularly acute when mappings are used within larger dataflow systems (Dessloch et al. 2008). For instance, we may need to infer mappings between dataflow nodes or would like to express mappings that depend on mappings defined earlier in the flow. Work on typed SQL combinators has helped alleviate similar challenges when exchanging purely relational data using Haskell (Leijen and Meijer 1999) and C♯ (Microsoft 2005–2006; Meijer et al. 2006).

Contributions

The primary idea behind this ongoing work is that schemas classify mapping expressions in much the same way that "types classify terms." Treating mappings as typed objects allows us to use classical type-theoretic and functional programming techniques to address the above challenges. Using this principle, we:

- Develop a formal theory of mapping polymorphism, including algorithms for type-checking mappings and inferring schemas from mapping expressions, and investigate polymorphism's connection to mapping systems, semantics, and re-use.
- Create a Haskell-ish, Trex-style (i.e. making essential use of extensible records, qualified types, and row-polymorphism (Gaster and Jones 1996)) domain-specific schema mapping language well-suited for "schema mapping in the large," implemented as an extension to Clio.

Highlights

Nested relational (NR) schema,

$$\begin{array}{rcl} Row & ::= & - \mid (\!| Row,\ \mathcal{L} : NR |\!) \\ NR & ::= & \texttt{ATOMIC}\ \mathcal{A} \mid \texttt{RCD}\ Row \mid \\ & & \texttt{SETRCD}\ Row \mid \texttt{CHC}\ Row \end{array}$$

describe the shape of data that consist of atomic elements, records, sets of records, and choices/variants/sums. We consider only rows without duplicate labels, and identify rows up to permutation. Example NR schema are

$$\begin{array}{lllll} \mathsf{Src} & :: & \texttt{RCD} (\!| & \mathsf{school} & : \mathsf{String}, \\ & & & \mathsf{date} & : \mathsf{String}, \\ & & & \mathsf{depts} & : \texttt{SETRCD} (\!| \mathsf{dept} : \mathsf{String} |\!) \ |\!) \\ \mathsf{Dst} & :: & \texttt{RCD} (\!| & \mathsf{projects} & : \texttt{SETRCD} (\!| \mathsf{projectId} : \mathsf{String}, \\ & & & & \quad \mathsf{taskId} : \mathsf{String} |\!) \ |\!) \end{array}$$

Constraints between instances of data conforming to NR schema are captured by *nested mapping expressions*. These expressions generate queries that materialize target instances satisfying the constraints. Mapping expressions resemble formuale of set-theory, like

$$\texttt{forall}\ p\ \texttt{in}\ \mathsf{Src.depts}\ \texttt{exists}\ q\ \texttt{in}\ \mathsf{Dst.projects}$$
$$\texttt{s.t.}\ \mathsf{Src.school} = q.\mathsf{projectId} \wedge p.\mathsf{dept} = q.\mathsf{taskId}$$

but with syntactic restrictions that ensure solutions can always be computed.

A distinguishing feature of the mapping language is its CHC eliminator, which has a single fixed branch. Supposing that $e :: \texttt{CHC}\ (\!| r, \mathsf{l} : t |\!)$, elimination of CHC is done with a binding construct $v\ \texttt{of}\ l\ \texttt{from}\ e.\ \phi(v)$, where $\phi(v)$ is typed assuming $v :: t$. (This behavior differs from many programming languages where choice elimination is required to have as many branches as choices.) Sets of mapping expressions may thus be required to specify constraints between variants.

Schema as types

The intuitive typing discipline of the mapping language (loosly speaking, as given by a naive encoding into O'Caml or Trex / Hugs / Haskell) guarantees the meaning of sets of mapping expressions as satisfiable constraints. For us, and for Clio, a *mapping* is a set of mapping expressions M and a source and target schema S, T for which $\vdash M :: S \to T$.

The principal type of M in this discipline can be expressed in the NR schema language extended with qualifiers and row (ρ) and schema (σ) variables, following the tradition of qualified types. The qualifiers enable row polymorphism and enforce a restriction that equalities must be between atomic schema elements. An example extended schema is

$$\texttt{atomic?}\ \sigma,\ \rho\ \texttt{lacks?}\ \mathsf{l} \Rightarrow \texttt{RCD}\ (\!| \rho, \mathsf{l} : \sigma |\!)$$

We can infer the unique principal satisfiable types of a set of mapping expressions by employing the complete row unification algorithm of (Gaster and Jones 1996). The main technical issue here is that we need to unify (pre-)row expressions like

$$(\!| \mathsf{l}_1 : t_1,\ \mathsf{l}_2 : t_2 |\!) \sim (\!| \mathsf{l}_2 : t_2,\ \mathsf{l}_1 : t_1 |\!)$$

but traditional unification distinguishes these permutations. The inference algorithm decides mapping expression set satisfiability.

Implementation

Using qualified types allows us to embed the mapping language into a λ-calculus closely resembling (Gaster and Jones 1996). The result of the embedding is an intermediate form usable by mapping engines for representing programs over mappings. A simple use of the form is to express mappings that depend on other mappings, which, for instance, occur in dataflow graphs of mappings (Dessloch et al. 2008). Our implementation, which automatically populates (adds arcs to) mapping graph skeletons (nodes are schema and arcs are mapping expressions) using schema-as-types techniques, represents updates to mapping graphs as programs.

Semantics

Principal types let us investigate the semantics of sets of mapping expressions M. Given a mapping meaning function $[\![(M, S, T)]\!]$ (e.g. taking mappings to queries), we can define a simple meaning for M as $[\![(M, P_S, P_T)]\!]$ where the P are concretizations of M's principal type. In practice, meanings defined in this way are most often enountered when integrating mappings with other systems – for instance, when mapping language embeddings are used by query generators.

Further type-directed analysis sheds light on mapping expression reuse. For instance, it is useful to automatically rewrite mapping expressions to *lift* them to apply to a "larger" schema, as in the example of reusing $M :: S \to Book$ as $M' :: S \to \texttt{RCD}\ (\!| \mathsf{book} : Book\ ,\ \mathsf{loanedTo} : \mathsf{String} |\!)$. Re-use of mapping expressions in this way can be studied as type coercion, and we have investigated lifting in particular.

References

Angela Bonifati, Elaine Qing Chang, Terence Ho, Laks V. S. Lakshmanan, and Rachel Pottinger. HePToX: Marrying XML and Heterogeneity in Your P2P Databases. In *VLDB(demo)*, pages 1267–1270, 2005.

S. Dessloch, M. A Hernández, R. Wisnesky, A. Radwan, and J. Zhou. Orchid: Integrating Schema Mapping and ETL. In *ICDE*, pages 1307–1316, 2008.

R. Fagin, P. G. Kolaitis, and L. Popa. Data exchange: getting to the core. In *PODS*, pages 90–101, 2003.

R. Fagin, P. G. Kolaitis, L. Popa, and W. Tan. Composing Schema Mappings: Second-Order Dependencies to the Rescue. *TODS*, 30(4):994–1055, 2005.

Ronald Fagin, Phokion G. Kolaitis, Lucian Popa, and Wang Chiew Tan. Quasi-inverses of schema mappings. In *PODS*, pages 123–132, 2007.

Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical Report Technical report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, November 1996. URL `http://www.cse.ogi.edu/~mpj/pubs/96-3.ps.gz`.

L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth. Clio Grows Up: From Research Prototype to Industrial Tool. In *SIGMOD*, pages 805–810, 2005.

Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*, volume 35, pages 109–122, New York, NY, USA, January 1999. ACM Press. doi: 10.1145/331960.331977. URL `http://portal.acm.org/citation.cfm?id=331977`.

Erik Meijer, Brian Beckman, and Gavin M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD*, page 706, 2006.

S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: A programming platform for generic model management. In *SIGMOD*, pages 193–204, 2003.

S. Melnik, P. A. Bernstein, A. Halevy, and E. Rahm. Supporting Executable Mappings in Model Management. In *SIGMOD*, pages 167–178, 2005.

Microsoft. Micorsoft Corp. The LINQ Project, 2005–2006. http://msdn.microsoft.com/netframework/future/linq/.

Renée J. Miller, Laura M. Haas, and Mauricio A. Hernández. Schema Mapping as Query Discovery. In *VLDB*, pages 77–88, 2000.

L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin. Translating Web Data. In *VLDB*, pages 598–609, 2002.