AFRL-RI-RS-TR-2019-075

SIRIUS: A TOOLSET FOR BUILDING FIRST-CLASS DOMAIN-SPECIFIC LANGUAGES

TUFTS UNIVERSITY

MARCH 2019

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2019-075 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
STEVEN DRAGER
Work Unit Manager

/ S /
QING WU
Technical Advisor, Computing
and Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MARCH 2019	FINAL TECHNICAL REPORT	DEC 2014 – OCT 2018

4. TITLE AND SUBTITLE

SIRIUS: A TOOLSET FOR BUILDING FIRST-CLASS DOMAIN-SPECIFIC LANGUAGES

5a. CONTRACT NUMBER
FA8750-15-2-0033

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

Samuel Guyer

5d. PROJECT NUMBER
MUSE

5e. TASK NUMBER
TU

5f. WORK UNIT NUMBER
FT

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Tufts University
161 College Ave.
Medford, MA 02155

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

DARPA
675 North Randolph St.
Arlington, VA 22203-2114

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2019-075

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Domain-Specific Languages (DSLs) have the potential to make it dramatically easier to produce high-quality software in a timely and cost-effective manner. This potential has been difficult to realize, however, because DSLs require so much work to build. One way to mitigate this cost is to embed the DSL in a general-purpose host language. Embedded DSL programs are compiled down to the host language, where a complete suite of tools already exists. The problem with this strategy is that the host tools work at the host language's level of abstraction, essentially forcing the programmer to perform tasks such as debugging and profiling on the implementation of the DSL, which is often totally unrecognizable. The goal of this project was to develop a set of techniques and tools that make it easier for DSL designers to build first-class domain-specific languages, which come equipped with a full suite of support tools that operate at the level of abstraction of the domain. Users of these DSLs will get the productivity and code quality benefits of DSLs throughout the development lifecycle, from editing and compiling to debugging and profiling. Our approach uses an embedding strategy in order to continue to obtain other benefits from the host language, including general-purpose programming, access to existing libraries, and the possibility of employing multiple embedded DSLs within a single application.

15. SUBJECT TERMS
Programming languages, compilers, domain-specific languages, semantics

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**STEVEN DRAGER**
U	U	U	UU	26	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

Table of Contents

1.0 SUMMARY

Domain-Specific Languages (DSLs) have the potential to make it dramatically easier to produce high-quality software in a timely and cost-effective manner. This potential has been difficult to realize, however, because DSLs require so much work to build. One way to mitigate this cost is to embed the DSL in a general-purpose host language. Embedded DSL programs are compiled down to the host language, where a complete suite of tools already exists. The problem with this strategy is that the host tools work at the host language's level of abstraction, essentially forcing the programmer to perform tasks such as debugging and profiling on the implementation of the DSL, which is often totally unrecognizable.

The goal of this project was to develop a set of techniques and tools that make it easy for DSL designers to build first-class domain-specific languages, which come equipped with a full suite of support tools that operate at the level of abstraction of the domain. Users of these DSLs get the productivity and code quality benefits of DSLs throughout the development lifecycle, from editing and compiling to debugging and profiling. Our approach uses an embedding strategy in order to continue to obtain other benefits from the host language, including general-purpose programming, access to existing libraries, and the possibility of employing multiple embedded DSLs within a single application.

Prior work has addressed some of the components of this vision, but not in an integrated fashion that supports the full process of developing an industrial-strength DSL implementation. Systems exist, for example, for specifying the syntax and semantics of new programming languages. The primary purpose of these systems, however, is to enable formal analysis of the properties of the language using tools such as Coq and Isabelle, not to serve as the basis for the language implementation. Our strategy was to integrate these techniques into a single DSL specification system that enables the automatic generation of a complete language implementation and associated tool suite.

Technical Approach

Our guiding vision is an integrated system for building the full suite of tools required to support a domain-specific programming language. Some parts of this vision require significant new innovations. In particular, there has been very little prior work on debugging and profiling support for DSLs. Our central insight is that the semantics of the language provide exactly the mapping we need between the high level DSL constructs and the underlying implementation. Operational semantics is a natural choice because it defines the meaning of a DSL program in terms of a sequence of computational steps on an abstract machine. Debugging operations such as "step" and "break" would map in a natural way to operations on the abstract machine. Figure 1 shows our system diagram.

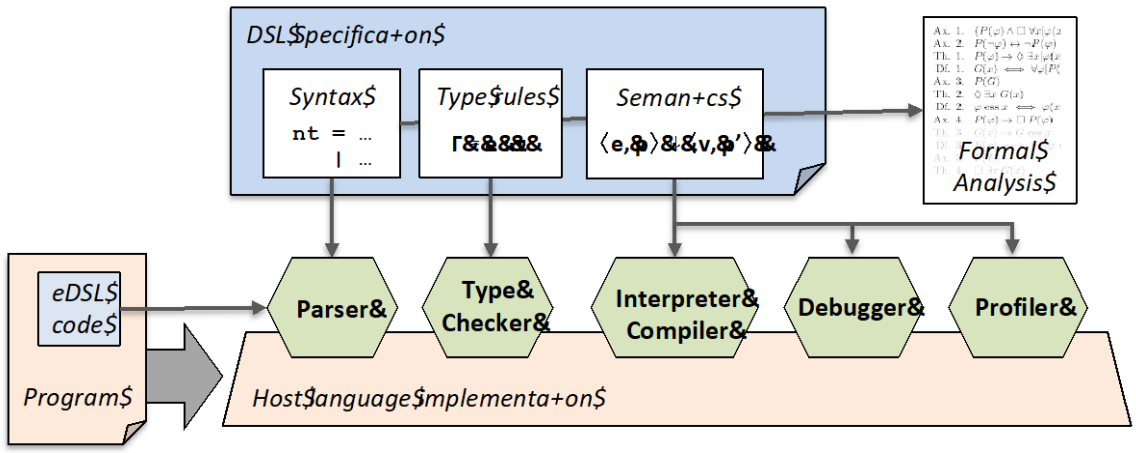

Figure 1: Sirius system diagram

Our approach uses an embedding strategy in order to continue to obtain other benefits from the host language, including general-purpose programming, access to existing libraries, and the possibility of employing multiple embedded DSLs within a single application. Initially, our work uses Haskell as the host language because it already provides some support for embedding DSLs. Our goal, however, is to develop a system design that can be readily incorporated into other general-purpose host languages and development environments. One of our main contributions is identifying the key features that a host language must implement to support our system. For example, we plan to design a concise Application Programming Interface (API) that the host language debugger needs to implement to enable DSL debugging.

Comparison with Current Technology

Prior work has addressed some of the components of this vision, but not in an integrated fashion that supports the full process of developing an industrial-strength DSL. Systems exist, for example, for specifying the syntax and semantics of new programming languages. The primary purpose of these systems, however, is to enable formal analysis of the properties of the language using tools such as Coq and Isabelle, not to serve as the basis for the language implementation. Other systems provide practical components for parsing and code generation, but stop short of supporting the full lifecycle, including optimization, profiling, and debugging. Our strategy is to integrate these techniques into a single DSL specification system that enables the automatic generation of a complete language implementation and associated tool suite.

Conclusions

Over the course of the project, we focused on two broad areas of investigation: (1) examples of domain-specific languages for several non-trivial domains, and (2) general tools and techniques for specifying, analyzing, and compiling these languages.

While we made considerable progress in both areas, we were unable to reach our ultimate goal of a complete, integrated system. Both pragmatic and theoretical challenges hampered us, and our experience suggests that such a goal is a much longer-term endeavor.

2.0 INTRODUCTION

Software pervades modern life, running on everything from traditional computers, tablets, and smart phones to routers, automobiles, and pacemakers to a vast array of military systems. Despite this success, there are still many hurdles to producing high-quality software in a time- and cost-effective manner [6].

Domain-specific programming languages promise to reduce these challenges by providing environments tailored to solving particular problems. General-purpose languages such as Java or C++ are intended to be suitable for almost any programming task and hence are beautifully suited to solving only a few. In contrast, each domain-specific language is narrowly focused on a particular domain. From this specificity arises its strength: The language, its syntax, semantics, compiler, and surrounding infrastructure can leverage domain knowledge to do more for the programmer than generic tools. Advantages include 1) raising the level of abstraction provided by the language, 2) enabling domain-specific program analysis and optimizations, and 3) enabling the generation of multiple artifacts from a single code base.

Raising the level of abstraction enables domain experts to write useful programs on their own, without requiring the expertise of skilled programmers [2, 12, 13]. Having specialized constructs in the language also allows programmers at all skill levels to write better code more quickly because the compiler can generate the boilerplate code that programmers would have had to write by hand in a general language [15, 7].

As a simple example, consider the problem of debugging a yacc-generated parser in C. The embedded DSL is the grammar specification, which describes productions and actions in the traditional Backus-Naur Form (BNF) notation. The implementation, however, consists of a single function called yyparse() with a giant switch statement that implements the table-driven representation of the Look-Ahead Left-to-Right (LALR) Deterministic Finite Automaton (DFA). It is impossible to debug the grammar by stepping through this function, in part because it requires a deep understanding of the Left-to-Right (LR) parsing algorithm, but more importantly because the DFA states are encoded as numbers whose meaning is completely opaque. In a similar way, traditional profiling would be useless because it would simply report that all time is spent in a single function called yyparse. The programmer cannot ask questions like “How much time is spent parsing this particular construct?”

DSLs can enable domain-specific program analyses and optimizations that are not available to general-purpose languages. For example, Structured Query Language (SQL) leverages relational algebra and the fact the language is not Turing complete to provide fast query processing. SpiralGen [15] makes use of algebraic techniques to produce implementations for Fast Fourier Transform (FFT) and software defined radio that are faster than state-of-the-art hand-written codes. Programs written in Cryptol [3], a DSL for writing cryptographic algorithms, can be analyzed to

determine that they require finite space and so can be implemented directly in hardware.

Finally, DSLs often can generate multiple artifacts from a single program. For example, the Parser for Ad-hoc Data Sets (PADS) [5] compiler can convert a single data format specification into a parser, a pretty-printer, and a variety of customized data analysis tools. The Event-driven State-machines Programming (ESP) [9] language can generate both a device driver and a proof of its correctness by calling out to a model checker. These advantages collectively mean that DSLs have the potential to reduce labor shortages, cost and schedule overruns, defect rates, and security vulnerabilities. As a consequence, DSLs are appearing in increasing numbers [16].

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

The goal of this project was make it easy to build first-class embedded domain-specific languages. Towards this goal we investigated techniques to specify all of the parts of a new Embedded Domain Specific Language (EDSL), including the syntax, type system, and semantics. A key objective is to come up with a unified specification that can be easily analyzed and manipulated to produce high-performance code. The most important pieces of the work are (a) the specifications (how to concisely collect the information we need) and (b) the generation mechanisms (how to produce all of the parts of the language implementation and associated tool suite).

Syntax

Our initial work on Sirius has not focused on surface syntax, since this area has been thoroughly investigated in prior work. We adopted standard technologies for specifying syntax, extended Backus-Naur Form, and the associated tools for generating parsers.

Type System

Our work on specific DSLs suggests that their type systems are not nearly as rich or complex as that of the host language. Ultimately, our goal is to support arbitrary type systems, but the work done so far does not require significant new capabilities. One area that we investigated in more detail is how the DSL type system interacts with the host type system, particular in cases where they are quite different.

Dynamic Semantics

One focus area of our research on this project was specifying dynamic semantics – essentially, describing what the DSL constructs "mean" in terms of the underlying host language. Our research started with so-called "big-step" semantics, which are easy for language designers to understand and specify.

In addition, big-step semantics provide a nearly automatic way to generate an interpreter for the DSL – an incredibly useful tool for investigating and testing the properties of the new language. From there, we investigated how to translate big-step semantics into small-step semantics, which are closer to the target host language. This component corresponds more closely to a compiler for the DSL, which is ultimately what the DSL designer provides to end users. This part of the project proved technically challenging and generated a number of interesting insights.

Debugger and Profiler

Source-level debuggers and profilers are challenging tools to build for any language, and are often completely absent in DSL implementations. The typical EDSL solution – relying on the host language tools – is not much better because it is ignorant of the domain-specific semantics, exposing the programmer directly to the low-level implementation of DSL constructs. During the course of this project we were not able to investigate the more general problem of generating these tools for any DSL. However, we made significant progress on tools for specific languages, including a general-purpose tool for debugging parts of the runtime system of a new language.

Generating a profiler presents some unique challenges, because the generated code must be sufficiently performant for profiling information to be useful. On the other hand, instrumenting the code for profiling is more straightforward than doing so for debugging, since it is not interactive. The essential idea is to associate timers and cost metrics with semantic rules, and instrument the associated code appropriately.

In the course of this project we also investigated a new way to do performance tuning using genetic algorithms. This work is described in more detail later.

4.0 RESULTS AND DISCUSSION

4.1 Language Tools

Our work on fundamental programming language tools falls into two categories: (1) work on the foundations of semantic representation, and (2) automatic performance tuning. These two sub-goals are described in more detail below.

4.1.1 Semantics

One major thrust of our research has been on general language tools for specifying, analyzing, and implementing domain-specific languages. Prior work has provided fairly complete solutions for both surface syntax (what the programmer writes) and type systems. We have focused on a major missing piece of this problem: connecting the syntax and types to the semantics of the language constructs. In particular, we provide a systematic way to specify the meaning of a language construct using a standard, formal notation called operational semantics. Operational semantics come in two general forms: big-step and small-step. Big-step semantics, as the name implies, describes the steps necessary to evaluate a language construct in fairly coarse terms. For example, evaluating an arithmetic expression involves evaluating the left and right sides to get two numbers, then applying the arithmetic operator. Small-step semantics describe the precise, low-level sequence of computations necessary to accomplish the big-step. Big-step semantics are nice because they are easier to specify and analyze, and they naturally correspond to the operation of an interpreter – in fact, an interpreter can often be generated automatically from a big-step semantics. Small-step semantics correspond more closely to the output of a compiler, enabling high performance. A key open question that we worked on in this project is whether a small-step semantics can be automatically inferred from big-step semantics.

Work performed

We started by developing a formalism and a prototype implementation to allow users to define an EDSL. The formalism is based on small-step operational semantics, which allows more detailed specifications, reasoning on a finer level, as well as generation of debuggers. We explored existing approaches allowing transformation of specifications into the more efficient big-step style semantics for running EDSL programs.

While existing approaches allow transforming a small-step into a big-step style semantics, the reverse direction has not yet been explored. We developed a systematic approach for transforming big-step into small-step, which enables our framework to accept both styles. We are developing an automatic transformer based on our advances that should integrate into the Siriusly toolset.

Initially, the development of a prototype transformer from big-step to small-step evaluators was implemented as a series of relatively simple transformations. Although they are mostly known and explored in the literature (transformation into continuation-passing style, defunctionalization), they had to be adapted to our

setting. We implemented a partial automatic evaluator transformer, which guided our subsequent formalization in an interactive theorem prover. A mechanically verified version of this transformer accompanies and supports the published version of our results, and the prototype was tested with small examples of call-by-value and call-by-name lambda calculi with extensions, and small imperative ("while") languages. Most phases in the prototype transformer have been implemented and tested on small examples, but progress in finalizing the prototype proved to be more difficult than anticipated.

Subsequent research allowed us to complete implementation of a transformer from deterministic big-step evaluators to small-step counterparts. The input is a canonical big-step interpreter in a functional language. The output is a small-step interpreter in a similar style. Moreover, both types of interpreters can be pretty printed as inference rules in LaTeX. The small-step semantics produced is usually close to what a person would write down, modulo some auxiliary constructs that can sometimes be expressed by combinations of constructs already in the language. We took the initial steps in formalizing the transformation in Coq, and wrote up these results in a paper submitted to the European Symposium on Programming (ESOP).

Subsequent work expanded and extended this framework with a series of features:

- Coq formalization: experimenting and reasoning about a locally nameless representation. Proved correctness of lambda lifting, Cyber Physical System (CPS), and continuation generalization.
- Implemented transformations for interpreters that handle state, such as memory.
- Implemented simplifications and optimizations on interpreter code that result in a more straightforward small-step semantics.
- Implemented step unfolding and empty continuation elimination.
- Experiment with styles of specifying exceptions in big-step semantics (as a value, as a state) and thought about approaches to get a corresponding standard small-step specification.

Results

The primary result of this thrust of the project is a new framework for transforming big-step semantics automatically into small-step semantics. This framework provides the theoretic foundation for a system in which language designers can specify their DSLs in an intuitive form (big-step semantics), but compile down to an efficient form (small-step semantics), while guaranteeing that semantics are preserved. A mechanized proof was developed to ensure that this transformation is correct.

4.1.2 Performance tuning

This project funded an undergraduate student (Remy Wang) during the summer exploring a new way to optimize programs written using lazy functional languages, such as Haskell. This work grew out of a class project done by a PhD student, Diogenes Nunez (funded on a different grant). Lazy languages often include an annotation that the programmer can use to force immediate (strict) evaluation of an expression. Strictness can dramatically improve performance when used in the right places, but it is notoriously difficult to figure out where. In some cases, adding a strictness annotation can cause a program to fail or to terminate. In this project we developed a genetic algorithm to select expressions for strictness. The fitness function is straightforward: programs that run faster are better, and any program that runs longer than the original (fully lazy) program are discarded.

Work performed

Undergraduate Remy Wang worked on an automatic method for inferring strictness annotations for Haskell programs. Programmers can use strictness annotations to tell the Haskell compiler where to use eager evaluation instead of lazy evaluation, which can have a dramatic effect on program performance, but is notoriously difficult to figure out.

The key idea in this work is to use a genetic algorithm to figure out where to place strictness annotations (i.e., where in the program to place eager evaluation operations). In our approach, programmers write their Haskell program without worrying about strictness annotations. Once they are happy with the correctness of their code, they run AUTOBAHN, supplying the program and representative data. AUTOBAHN uses a genetic algorithm to search through the space of all possible bang patterns to find candidate annotations that reduce the value of a fitness function selected to improve program performance. AUTOBAHN can start with a program that already contains bang patterns or one that does not. It has the power to both add and remove annotations. AUTOBAHN returns a list of annotation sets, ranked by a measurement of how much each annotation set improved performance. Programmers examine the proposed alternatives for soundness on relevant program inputs and decide whether to have AUTOBAHN produce modified sources corresponding to one of the generated annotation sets.

The genetic algorithm iteratively considers a collection of candidate annotations. In each round, it preserves those annotations that demonstrate the best performance on the supplied data. Since AUTOBAHN starts with the original program, AUTOBAHN is guaranteed to only suggest alternative annotations that actually improve the original performance on the supplied dataset.

Results

We showed that genetic algorithms can be used to automatically infer strictness annotations that enable non-expert Haskell programmers to improve the performance of their programs on a variety of different performance criteria: total runtime, garbage collection time, and live size (aka, peak allocation).

We demonstrated the effectiveness of this approach on 60 programs from the NoFib benchmark suite, showing geometric mean improvements of 8.5%, 18%, and 7.2%, and maximum improvements of 89%, 98%, and 99.3% on the total runtime, garbage collection time, and live size performance criteria, respectively.

We used AUTOBAHN in a case study to optimize the performance of a garbage collector simulator gcSimulator. The annotations inferred on a small training set resulted in performance improvements on larger data sets: 23.6% decrease in running time and a reduction in live size to under 1% of the unoptimized program on the full dataset.

We showed in a second case study that AUTOBAHN can infer application-specific annotations for the Aeson library code, optimize driver programs, validate and convert different annotations to produce optimal behavior.

We conducted 10-fold cross-validation studies for gcSimulator and convert, showing that the inferred annotations are stable across different data sets. For gcSimulator, the study also shows that the inferred annotations generally outperform the annotations added by hand by the original author.

4.2 Domain-specific Languages

One of the key strategies in this project was to guide our design of general-purpose language tools by developing several real DSLs for specific domains. This work both informed our choices and provided real-life tests that are not contrived or tailored to our specific system. These two languages are:

- Warble: a DSL for programming wearable devices based on microcontrollers, sensors, and actuators (Internet of Things (IoT) systems).
- Floorplan: a DSL for specifying memory managers and garbage collectors. A secondary benefit of Floorplan is that it dovetails nicely with our overall goal of providing more complete support for language implementation. Using Floorplan a language designer can build sophisticated memory managers for their languages with much less effort.

4.2.1 A DSL for wearable devices

Graduate student Matt Arhens has been working on a higher-order language, called Warble, for programming distributed sets of microcontrollers (e.g., Raspberry Pi or Arduino). Warble is specifically targeted at teaching students about engineering, so it pushes the limits of our framework in making languages easy to specify and easy to use by programmers. Warble was initially developed as an EDSL inside Haskell, allowing us to use the tools developed in the more general language work. The main challenges in developing Warble are a result of its highly concurrent nature, and its explicit notion of time and timing. In addition, the target hardware (for example, Arduinos) is not very powerful and extremely memory-constrained.

Work performed

An initial prototype of the language was explored with a group of high school students. A procedure was developed for teaching non-technical users general purpose programming in Haskell. The evaluation tested the traditional methods in which a language designer would create an embedded DSL in Haskell and iterated with users. The lessons learned, specifically around iterating on a concrete syntax, abstract syntax, runtime system, and supporting standard library simultaneously, proved useful to the design of the general purpose tools as many other language design workbenches focus on ”complete” languages that are designed up front and simply need to be implemented or languages so small that they do not necessitate iteration.

Warble improved by use case feedback that influenced new expressions and type checking constructs. Specifically constructs about well- timed embedded devices and the need for a flexible relationship between static analysis – what wearable programs are invalid for a configuration of hardware – and dynamic analysis – how should the wearable program react to effects that cause hardware to violate the timing contracts / types. Many of the other EDSLs used as use cases and tooling do not make use of a dependent type system, which should introduce interesting host language – EDSL – target language interaction requirements Siriusly tools should facilitate.

Warble went through a second iteration that incorporated feedback from the user study, with expectation of a second user study spring 2017. The language gained more powerful constructs, rather than relying on library members being written in the host language to meet the aforementioned requirements around timing and error. In addition to generating Haskell source, targeting a smaller device friendly target language, such as a subset of C, necessitated major changes to the underlying runtime system as well, in the hopes of making it more modular: developing one runtime feature of the language at a time as small services to better promote iterative development.

The Warble framework was broken down into smaller sub-expression languages: Signal and Timing, Transform and flow control, composition, simulation, query, and executable – each with a corresponding interface for a runtime system service in Haskell and a C subset. As a precursor to the Haskell and C subset languages, we developed an intermediary form at the value level, which provides the portable representation of the program for both execution on hardware and simulation. This core language was defined from principles that we are expecting out of Siriusly compatible DSLs such as a core calculus made up of a minimal Abstract Syntax Tree (AST) definition and operational semantic rules for evaluation. Syntactic sugar and other niceties (e.g. EDSL language extensions) are also included as optional syntactic forms and rules that evaluate to the core calculus for inclusion in the Siriusly model.

Subsequent work expanded the features of Warble and built out the toolchain:

- We developed a set of representative examples, both full example programs (greenhouse program, laser tag) and micro benchmarks (code coverage of expressions) implemented for the logical circuit portion of Warble.
- We developed the core eval function for the Haskell implementation, in addition to the compiler and a small Virtual Machine (VM) for embedded devices.
- We wrote formal type rules for the linear circuit case (no branches), and operational semantics for the linear circuit case (no branches).
- We started on proofs of behavior and elimination of timing and data conversion bugs using Warble instead of low-level imperative C code directly, program equivalence proofs, and full example programs (greenhouse program, laser tag).
- Designed and implemented a rate robustness analysis (with an arbitrary time unit instead of mandating milliseconds) and speedup / less complexity of type checking of numeric rate types by using Template Haskell metaprogramming instead of leveraging the Haskell type system using Haskell language extensions. The rate checker delegates the proof to the Z3 Satisfiability Modulo Theories (SMT) solver.

Towards the end of the project, we specified the value type for Warble that described an abstract machine description rather than a numeric value or opaque block Input-Output (IO Action). We encode this description as an abstract syntactic form and compare to the current dynamic semantics (partial implementation). We implemented this value form in the prototype implementation of the EDSL Warble. It is now easier to recognize the value as a machine (a set of inputs, outputs, setup function, loop forever function, and regular delay interval) in the implementation. The separation of the Warble language, the Host implementation (compiler) language and the Target (generated code) language is now clearer in both the design and implementation of Warble. This benefit can generate to other EDSLs that describe computation or code as output values rather than typical data.

Results

Work on Warble proved challenging, both as a DSL and as a test for the Siriusly framework. The main challenges included the use of complex types, timing-related semantics, and the goal of targeting students as users. The resulting language has grown and matured significantly, and we hope to deploy a complete version soon. We also used Warble to explore the deep embedding vs shallow embedding for EDSLs.

4.2.2 A DSL for memory managers

In modern runtime systems, memory layout calculations are hand-coded in low-level systems languages. Unfortunately, low-level language primitives like structs are not powerful enough to describe a rich set of layouts. Even worse, implementing grungy pointer arithmetic, repetitive offset constants, and bitmaps is a tedious and error-prone process. Furthermore, the necessity of fine-grained control over algorithmic runtime complexity inhibits automated bug detection and algorithm analysis.

In this part of the project we developed Floorplan, a declarative language for specifying memory layouts at a high level. Floorplan provides a precise specification language for describing constraints on the layout of memory, founded on logical multiples: a novel application of regular expressions augmented with an existential quantifier. Floorplan also formalizes the description of various memory layout idioms performed by real-world memory managers. Finally, Floorplan is implemented as a compiler for generating a Rust library from a Floorplan specification, with core memory layout calculations proved correct by machine-checked proofs in Coq.

This work supports the dual goals of testing our Siriusly design framework and providing better support for language designers by giving them an easier way to build runtime systems.

Work performed

Initially, we built a prototype of the PADS parsing system retargeted for in-memory structures. The main observation is that the heap has a well-defined structure that is dictated by the underlying memory manager. It includes information such as how individual objects are laid out, how free and used memory blocks are managed (e.g., free lists vs bitmaps), and how the system divides up the global address space (e.g., different regions for different object lifetimes). In most cases, this architecture is not represented explicitly anywhere in the implementation – at best, in comments describing it. In addition, these systems are very hard to debug because they are so low level. Our idea is to use PADS to write an explicit specification of a properly formed heap in terms of a set of high level types. The PADS system can then parse memory to determine if the structure and invariants have been respected. This approach represents a big step forward in helping to build reliable runtime systems.

The primary student on the project, Karl Cronburg, also developed an implementation of his memory manager debugger, called PermChecker. PermChecker is a tool that plugs into the Pin dynamic binary rewriting system, which allows us to use the tool with almost any language runtime. We are currently configuring it to debug the JikesRVM Java virtual machine, which is an extremely challenging target because it is implemented in Java itself (self-hosting), and it generates optimized code on-the-fly.

Karl Cronburg shifted his focus from heap parsing using PADS to an efficient online checking mechanism based on the Pin binary instrumentation tool. The idea is to directly model the state of each piece of memory as it moves through the memory management system. Each component of this system is responsible for managing blocks of memory at a particular granularity. It receives memory from the layer below, carves it up, and hands it out to the layer above. A sophisticated runtime can have as many as five or six different layers, starting with the Operating System (OS) at the bottom up to the application at the top. Our tool, called PermChecker, makes sure that a given layer only touches the memory that it is currently responsible for – we model this idea as permissions. Errors in the memory manager show up quickly as permission errors, rather than as confusing memory corruption errors later in the execution.

The first phase was to look at concrete prototypes of memory layouts in various memory allocators and garbage collectors using more than one framework. First, we focused on the manner in which researchers present their memory abstractions in the papers they write. The argument for this choice is that to prototype an abstraction language to be used by researchers, we must model the abstractions they are creating in a form closely matching the natural way in which they think about and present these abstractions to other researchers. More concretely, the goal of such a memory abstraction language is to make the memory abstractions researchers think in terms of (blocks, objects, cells, heaps, pages, pointers, ...) first-class values that we can all reason about. Our initial framework modeled memory layout abstractions as a pyramid structure, where allocating and deallocating adds layers / levels to the pyramid. Each layer corresponds to a type of memory abstraction at which memory is allocated. This framework was useful for prototyping basic debugging capabilities (permissions on memory), but was found to be not expressive enough to model more complex memory allocation schemes. In our improved framework prototype, we are modeling memory abstractions as composable types using a grammar-like syntax. Namely a program written in an (Haskell EDSL) implementation of this framework in its surface syntax most closely corresponds to the possibly-recursive graph-like structure of a memory model for a particular runtime system implementation.

The second phase has been developing a declarative, domain-specific language, called Floorplan, for describing the structure of a heap as laid out by a memory manager. Floorplan is inspired by PADS [16], a language for describing ad-hoc data file formats. A Floorplan specification looks like a grammar, augmented with a number of features designed specifically for this domain. In particular, Floorplan

provides powerful ways to specify the sizes, alignments, and relationships among chunks of memory, resulting in very compact descriptions. The key idea is that any correct state of the heap can be represented as a string (a sequence of bytes or tokens) derived from a Floorplan grammar. Grammars are a natural choice because they match the configuration of most modern memory managers, which comprise layers of code that carve up memory into smaller and smaller pieces.

Floorplan does not, however, attempt to capture the policy details of any particular memory management algorithm. The closest Floorplan gets is in its ability to logically connect two or more pieces of memory. The Floorplan compiler provides the mechanisms by which an algorithm can be implemented in the form of offset constants, bit field accessors, array iterators, etc. In contrast the memory management algorithm itself comprises the policies by which memory gets allocated. For example the Floorplan compiler automates the synthesis of constants and pointer calculations for accessing an object liveness bitmap while saying nothing about spatial relationships among a set of live or reachable objects in a heap.

Results

This component of the project made the following contributions:

- A specification language founded on a novel application of regular expressions augmented with an existential quantifier construct, Logical Multiples (LMs).
- Formal specifications of various idiomatic memory layout concepts found across academic and industrial runtime systems, sufficiently powerful enough to implement the memory layout of a state-of-the-art garbage collection algorithm: immix.
- Soundness and completeness proofs of theorems involving pointer arithmetic calculations generated by the Floorplan compiler and machine-checked in Coq.
- A Floorplan compiler targeting Rust.
- An ad-hoc analysis of integrating a Floorplan spec with the Rust implementation of Immix.

4.3 Related Work

Prior work on tools for defining and extending programming languages is extensive, but tends to focus on particular aspects of the problem (such as syntax or types or semantics) rather than the full problem of end-to-end language implementation. Our strategy was built on prior work when possible, integrating it into a more complete system and filling in oft-neglected gaps, such as debuggers.

The Spoofax language workbench provides an environment for developing domain-specific languages from declarative specifications [8]. The focus of this work is primarily on front-end tools, such as parsers and plug-ins for Integrated Development Environment (IDEs), as well as on systems of rewrite rules for translating DSL code into host language code. The foundational construct for much

of this work is the abstract syntax for the language. Our goal was to build on these ideas by adding the ability to specify type rules and semantic rules that augment the abstract syntax with deeper information about what the DSL constructs mean. With this information we can support a wider range of capabilities, including formal analysis of the type system and automated reasoning about DSL programs.

Ott [14] (and to a lesser extent TinkerType [10]) is a system designed for writing formal definitions of programming languages and calculi in a light-weight American Standard Code for Information Interchange (ASCII) notation to enable formal study. Ott takes these definitions and produces a number of artifacts, including 1) a latex version of the syntax and specification so they can be typeset in the traditional style; 2) Coq, HOL, and Isabelle/Higher Order Logic (HOL) versions so designers can write machine-checked formal proofs; and 3) an Objective Categorical Abstract machine Language (OCaml) version of the syntax to help with an initial implementation. Although Ott is a suitable starting place for Sirius, it is not sufficient: Ott is focused on formal study rather than actual implementation and Ott does not attempt to define new languages as extensions to an existing one. We have explored using Ott to generate Latex, Coq, HOL, and Isabelle/HOL bindings for EDSLs written in Sirius.

PLT Redex [4] is a system for specifying and debugging operational semantics. It allows the user to specify the syntax of the language under study using a grammar and a set of reduction rules. From this input, PLT Redex provides tools to help the user understand and debug the semantics, including a way to interactively evaluate program terms and a random test case generator. Like Ott, PLT Redex is a suitable starting place for specifying the operational semantics of Sirius EDSLS, but it is not sufficient. PLT Redex is focused on formal study of the operational semantics, not using them as the basis of an implementation and PLT Redex does not directly support defining a new language as an extension to an existing one. Taking inspiration from PLT Redex, we studied using the Haskell QuickCheck [1] library to generate random test cases to help EDSL designers debug their operational semantics.

Wu and Gray describe a system for generating testing tools automatically for a domain-specific language [17]. While generating testing tools for EDSLs is not directly in the scope of this project, it is certainly worth considering as an extension.

There is little prior work on building debuggers for domain-specific languages (as distinct from debugging the DSL specification itself). Closely related to our goals is the work of Lindeman et al [11], which uses a declarative specification to generate a domain-specific language debugger in the Spoofax language workbench. Their debugger specification maps debugging operations, such as “step”, “enter”, and “exit”, to fragments of the abstract syntax, and describes how to instrument them. The downside of this approach is that it will not work well when the DSL itself is declarative, so that the execution model is not readily identifiable in the abstract syntax. Our hypothesis is that mapping debugging operations to steps in the operational semantics will accommodate a much wider range of DSLs.

Wu et al describe a similar system that implements a debugging interface between a domain-specific language and the Eclipse IDE debugger Java Development

Environment (JDE) [18]. This system interprets debugging operations, such as "step" and "break", by looking at the grammar for the DSL. For example, stepping over a particular construct in the DSL is implemented as a series of steps over the code that it generates. The system retains information about the DSL line numbers and variables names to provide a DSL view of the program state. This system integrates nicely with the host language debugger, and we plan to use a similar strategy of connecting to "hooks" in the debugger. Building solely on the grammar, however, some steps in the DSL may represent very large chunks of computation. We believe that using the language semantics provide a more meaningful connection between the high-level constructs and how they are implemented.

5.0 CONCLUSIONS

The goals of this project proved a significant challenge. In the process, however, we made significant progress and reached many important milestones:

- We designed a DSL for wearable electronics, called Warble, and built both a compiler for the language and a formal proof of correctness for the timing-related properties.
- We designed a DSL for specifying memory managers and garbage collectors, called Floorplan, and compiler that automatically generates high-performance code in Rust.
- We explored the issues in specifying language semantics using big-step operational semantics and developed a technique to transform them into small-step semantics.

We are committed to the overall objectives of the project and will continue to pursue our goals and build the system components that comprise a complete solution.

6.0 REFERENCES

[1] Claessen, K., and Hughes, J. Quickcheck: A lightweight tool for random testing of haskell programs. In Proceedings of the Fifth ACM SIGPLAN Inter- national Conference on Functional Programming (New York, NY, USA, 2000), ICFP '00, ACM, pp. 268–279.

[2] Cortes, C., Fisher, K., Pregibon, D., Rogers, A., and Smith, F. Hancock: A language for analyzing transactional data streams. TOPLAS 26, 2 (2004), 301–338.

[3] Cryptol. http://corp.galois.com/cryptol/.

[4] Felleisen, M., Findler, R. B., and Flatt, M. Semantics Engineering with PLT Redex. The MIT Press, 2009.

[5] Fisher, K., and Gruber, R. PADS: A domain specific language for processing ad hoc data. In PLDI (June 2005), pp. 295–304.

[6] Advancing Software-Intensive Systems Producibility; National Research Council, 2010, Critical Code: Software Producibility for Defense. The National Academies Press, 2010.

[7] Forest: A language and toolkit for programming with file system fragments. http://forestproj.org, 2010.

[8] Kats, L. C., and Visser, E. The spoofax language workbench: Rules for declarative specification of languages and ides. In Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (New York, NY, USA, 2010), OOPSLA '10, ACM, pp. 444– 463.

[9] Kumar, S., Mandelbaum, Y., Yu, X., and Li, K. ESP: A language for programmable devices. In Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (New York, NY, USA, 2001), PLDI '01, ACM, pp. 309–320.

[10] Levin, M. Y., and Pierce, B. C. Tinkertype: A language for playing with formal systems. J. Funct. Program. 13, 2 (Mar. 2003), 295–316.

[11] Lindeman, R. T., Kats, L. C., and Visser, E. Declaratively defining domain-specific language debuggers. In Proceedings of the 10th ACM Inter- national Conference on Generative Programming and Component Engineering (New York, NY, USA, 2011), GPCE '11, ACM, pp. 127–136.

[12] Matlab. http://www.mathworks.com/products/matlab/.

[13] R. http://www.r-project.org.

[14] Sewell, P., Nardelli, F. Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., and Strni˘sa, R. Ott: Effective tool support for the working semanticist. Journal of Functional Programming 20, 1 (2010), 71–122.

[15] Spiralgen. http://www.spiralgen.com.

[16] van Deursen, A., Klint, P., and Visser, J. Domain-Specific Languages: An Annotated Bibliography. SIGPLAN Not. 35, 6 (June 2000), 26–36.

[17] Wu, H., and Gray, J. Automated generation of testing tools for domain- specific languages. In Proceedings of the 20th IEEE/ACM International Con- ference on Automated Software Engineering (New York, NY, USA, 2005), ASE '05, ACM, pp. 436–439.

[18] Wu, H., Gray, J., and Mernik, M. Grammar-driven generation of domain- specific language debuggers. Software, Practice and Experience 38, 10 (2008), 1073–1103.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
AST	Abstract Syntax Tree
BNF	Backus-Naur Form
CPS	Cyber Physical System
DFA	Deterministic Finite Automaton
DSL	Domain Specific Language
EDSL	Embedded Domain Specific Language
ESOP	European Symposium on Programming
ESP	Event-driven State-machines Programming
FFT	Fast Fourier Transform
HOL	Higher Order Logic
IDE	Integrated Development Environment
IO	Input-Output
IoT	Internet of Things
JDE	Java Development Environment
LALR	Look-Ahead LR
LM	Logical Multiples
LR	Left-to-Right
OCaml	Objective Categorical Abstract Machine Language
OS	Operating System
PADS	Parser for Ad-hoc Data Sets
SMT	Satisfiability Modulo Theories
SQL	Structured Query Language
VM	Virtual Machine