Generative Programming for Embedded Systems
Final Report

Kevin Hammond*       Edwin Brady*

December 24, 2008

1 Overview
The aim of this project was to develop new generative programming technology for resource-sensitive systems, notably real-time embedded systems. Generative programming involves automatic generation of target code from some higher-level source under programmer control. It is especially suitable for implementing domain-specific languages, as deployed for the construction of embedded systems, for example, offering advantages of performance, tunability, flexibility and ease of implementation. The focus of the project has been on providing strong, type-based, and formally verifiable guarantees of the properties of generated code.

The project employed one post-doctoral researcher, Dr Edwin Brady, for a period of three years, and one post-doctoral research, Dr Christoph Herrmann, for a period of three months. These researchers benefited from the synergies offered by working as part of a large research team at the University of St Andrews, headed by the PI (Prof. Kevin Hammond), comprising two academics, four contract researchers and three research students. They also benefited from synergies with the EU-funded EmBounded project (Automatic Prediction of Resource Bounds for Embedded Systems), which has developed automatic time- and space-resource analyses for the Hume domain-specific language. Edwin now has a full-time academic position at St Andrews University, and Christoph is employed on an MOD-funded project in a related area. Both these researchers and the PI are full members of IFIP WG2.11 (Generative Programming), Edwin being elected on the basis of his work on this project. Both Edwin and the PI have also served on the programme committee for the ACM Conference on Generative Programming and Component Engineering (GPCE), and Kevin Hammond has also initiated a successful international workshop on Automatic Program Generation for Embedded Systems, with Prof. Paul Kelly (Imperial College).

All the outputs of the project can be found on the project web page, http://www-fp.cs.st-and.ac.uk/GPES.

1.1 Actual versus Planned Expenditure

<table>
<thead>
<tr>
<th>Category</th>
<th>Planned</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Staff</td>
<td>£90,264.00</td>
<td>£96,790.50</td>
</tr>
<tr>
<td>Travel</td>
<td>£20,421.00</td>
<td>£12,305.70</td>
</tr>
<tr>
<td>Consumables</td>
<td>£4,058.00</td>
<td>£3,095.72</td>
</tr>
<tr>
<td>Indirect Costs</td>
<td>£41,522.00</td>
<td>£44,523.62</td>
</tr>
<tr>
<td>Total</td>
<td>£156,264.00</td>
<td></td>
</tr>
</tbody>
</table>

The over-spend on salaries (and consequently indirect costs) was partially due to increases in salary costs as a result of university settlements over the three years of the grant, and partially because we employed Christoph Herrmann to work on the grant, contributing to the overall domain-specific language approach,
developing linkages with the SEAS and EmBounded projects, and exploring limitations of alternative, potentially lighter-weight approaches than the dependent type approach that we have adopted. We therefore did less travel to the USA than originally planned for this grant (this was, however, balanced by travel on other grants that we could not predict when applying for the grant), and spent slightly less on consumables.

1.2 Primary Research Achievements

In the course of this project we have explored generative programming techniques for designing, implementing and verifying resource sensitive systems. Our results show that dependently-typed frameworks provide a suitable representation for heap costs [6] and domain specific languages [7, 8]. We have also developed an extensible and embeddable theorem-proving library IVOR [3] and, using this, implemented a dependently-typed programming language IDRIS [4] and its compiler.

We have developed our dependent-type based approach, and our language implementation, by applying it to several examples related to embedded systems and hardware implementation. In addition to the extra-functional heap usage and resource management problems detailed in [6, 8], we have investigated the representation of functional properties of realistic problems such as binary arithmetic [12] as well as a more general investigation into the relative merits of dependent type systems and weaker functional language approaches [11]. We have thus made major progress in deploying advanced type-based approaches for generative programming in resource-sensitive contexts, as specified in the original research proposal.

1.3 Conformance to Original Research Objectives

Overall, the objectives of the project remained broadly aligned with those stated in the grant proposal. In particular, as planned, we developed generative programming approaches based on strong type systems and applied these to resource-related problems that are directly applicable to embedded systems. In developing our solutions, we benefitted from research carried out on the EmBounded project that has developed high-quality automatic time and space resource usage analyses that can be exploited by our higher-level type system. In order to better reflect Edwin Brady’s PhD expertise, and in line with topical research issues, the focus of the research shifted slightly to the use of dependent type based techniques for generative programming, and for proving both functional and extra-functional properties of programs.

Having developed dependent-type based approaches for program generation, we found it beneficial to develop a new dependently-typed language, IDRIS, based on a theorem-proving system IVOR (also developed in the course of this project) rather than modifying our existing Hume language [16]. By adopting this approach rather than the one outlined in our original methodology, we were free to consider how the dependent-type approach could be applied to the broader research problem, rather than being forced to solve some difficult technical problems which arise from adding a new type system to an already existing language (in this case, transferring dependent pattern-matching and implicit arguments into the Hume computation and co-ordination notations).

IDRIS provided an ideal framework for experimenting with type system implementation, primarily because using a dependent-type system makes it possible to embed domain-specific type systems within a host type system, in a similar manner to embedding a domain-specific language in a host language [17, 14]. We implemented a compiler for IDRIS, including some optimisations arising from our previous work [2, 20].

Although we have used a different language notation from that we originally intended (IDRIS rather than Hume), program generation techniques have remained central to the research we have carried out. In particular, we have clearly demonstrated that multi-stage programming (a form of generative programming) can be applied to dependent type systems [7]. The goal of multi-stage programming is to allow abstraction without the run-time overhead. One important abstraction is abstract syntax, e.g. of a domain-specific language. We developed several examples in our system, using the DSL approach, with direct application to embedded systems and hardware implementation.

We do believe our methods can ultimately be implemented in a modified version of Hume, by replacing the computation layer with the IVOR type theory. This is, however, purely a technical rather than a research problem.
2 Tools

In the course of the project we have implemented three interconnected tools to support our research.

- IVOR [3], an implementation of a dependent type theory with pattern matching and a tactic based theorem prover, accessible as a library from Haskell, and therefore extensible (by writing tactics using combinators accessible from Haskell) and embeddable (in Haskell applications). IVOR was originally intended as a core language for representing resource bounded programs in the style of [6], using Haskell to implement domain specific tactics for reasoning about costs. We have, however, also used IVOR to experiment with multi-stage programming [7] and as the core of a dependently typed programming language.

- IDRIS [4], a language with dependent types. IDRIS is built on top of IVOR, compiles via C and provides a foreign function interface for interacting with C. Being built on top of a theorem proving library allows easy interaction with the theorem proving tools, and a clean separation of programs and proofs. The foreign function interface allows experimentation with realistic programs executing low level operations such as memory allocation and concurrency.

- EPIC, a supercombinator language which compiles to executable code via C, usable as a back end for a functional programming language. We have used this as a compiler for IDRIS programs.

Each of these tools is available to download via the project web site.

3 Shallow embedding of a resource aware language

We began by investigating the representation of heap costs in typed languages [6], using the notion of sized types [18]. Since dependent types allow the embedding of values within types, we investigated the effects of a shallow embedding of the sized type system within a host dependently typed language, initially using COQ [13].

3.1 Representation of costs

The key idea behind our initial framework is that each user defined type is represented within the framework by a type predicated on a natural number, \( \mathbb{N} \). Thus we can embed size information explicitly within a type, and represent proofs of size properties directly in code. e.g. Given the user defined type of lists . . .

```haskell
data List a = nil | cons a (List a)
```

. . . we can create a “sized list” type in our dependently typed framework as follows, where the size of the empty list is 0, and the size of the non-empty list is one more than the size of its tail:

```haskell
data ListS : (A : *) \rightarrow \mathbb{N} \rightarrow * where
nilS : ListS A 0
consS : (x : A) \rightarrow (xs : ListS A xsn) \rightarrow ListS A (s xsn)
```

Note that the element type \( A \), like all types within the framework, is also predicated on a size. We use the convention that sized types (and their constructors) generated from the source language are given the suffix \( S \). We can be flexible as to what the size information for a structure is; whether it be high level information such as the above length of list, or the total size of all elements in the list, or more low level information such as the number of heap cells required to store a structure. Within our framework, the meaning of the size index of a family is not important, what matters is that the index satisfies the required properties.

To capture the effect of functions on size, we define the Size type which pairs a sized value with a predicate describing the properties that value respects:

```haskell
data Size : (A : \mathbb{N} \rightarrow *) \rightarrow (P : \forall n:\mathbb{N}. A n \rightarrow *) \rightarrow * where
size : (val : A n) \rightarrow (p : P n val) \rightarrow Size A P
```
Functions in the source, sized typed, language are translated to representations in the framework as a shallow embedding in the type theory. These representations take sized arguments, and return a Size structure. Within this framework, we were able to represent simple functions such as **append** (appending lists) and **split** (partitioning lists for example for sorting) but also, more interestingly, higher order functions such as **map** and **fold**. All of the sized constraints generated proof obligations, which were discharged using a Presburger arithmetic solver [21]. Embedding in a dependently typed metalanguage gave the following advantages:

- We could express more complex properties than those available in a sized type system, by lifting arbitrary functions into types.
- Where an inference system cannot derive a size recurrence, or where the user requires weaker constraints, we can allow the user to specify and verify a constraint by hand.
- If automated proof construction is not possible (e.g. the constraint is not a Presburger formula) we could expose proof obligations to the user.
- We could potentially use the type system to verify constraints given by an external inference system, e.g.[23]

With this initial investigation, we showed the value of representing constraints in a dependently typed host language, and continued with this approach throughout the project.

### 3.2 Ivor, a proof engine

We aimed to generate such shallow embeddings direct from a source language (related to Hume), but in doing so found the need for a tighter integration with the theorem prover. To some extent Coq’s tactical language allowed the construction of domain specific tactics for inserting and proving cost constraints but it was found difficult to achieve the automated construction of programs.

We regularly found it necessary to construct programs with *metavariables* — typed variables standing for parts of the program yet to be written — typically in place of proof obligations arising from programs.

We developed **IVOR** [3], an implementation of type theory, accessible as a Haskell library, which arose from Edwin’s PhD work [2]. The initial implementation included a type checker, a unification algorithm, some simple tactics and tactic combinators.

Implementing this library proved beneficial for several reasons:

- It became possible to build type theoretic representations of source programs easily.
- It gave a means of implementing relatively complex domain specific tactics.
- It allowed experimentation with the structure of the underlying type theory, especially the addition of *staging* annotations [22].

The first use of **IVOR** was an investigation into staged dependently typed programming [7]. The result of this investigation, that dependent types and staging annotations can co-exist, is important in that it allows us to use dependent types as a verification mechanism, and multi-stage programming as an efficient implementation technique.

### 3.3 Idris, a language with dependent types

Aiming for a source language tightly integrated with the theorem prover naturally led to an implementation of an experimental dependently typed functional programming language, **IDRIS** [4]. Although not initially an objective of the project, this language has proved a valuable tool for implementing and verifying resource aware languages. We have implemented a compiler for **IDRIS**, and added practical tools such as input/output facilities and concurrency.

An important feature of **IDRIS** is that it allows the user to separate programs and proof obligations which arise from this programs. Our primary interest is in implementing languages with resource aware
type systems, and with Idris we aim to represent these type systems as directly as possible. Such representations require proofs of constraints (e.g. memory usage validation) which arise naturally from programs, but are not part of the program. The type checker therefore allows the user to write programs which are neither complete or type-correct, but inserts *matavariables*, to be instantiated by proofs which will complete and correct the program.

### 3.4 Functional properties

While the primary focus of the project was to verify non-functional properties of embedded systems, the more general approach we have taken allows us to represent functional properties.

Our general approach is to reason about a simple representation (i.e. easy to reason about, but inefficient), then use dependent types to link this representation to a complex representation (i.e. efficient, but difficult to reason about). In [12], we apply this method to a hardware design problem, a binary adder. The simple representation we choose is unary natural numbers. It is easy to prove properties such as commutativity and associativity of addition with this representation, but obviously it is unsuitable for implementation in hardware. We gave a representation of binary numbers index by their unary equivalent, and in doing so proved that binary addition was equivalent to the verified unary implementation. In [4] we showed how the improved notation in Idris meant that the necessary proof obligations for verification of binary addition (properties of unary natural numbers) could be generated automatically.

### 4 Resource aware domain specific languages

While we have found the shallow embedding of heap costs to be effective, this approach has its limitations. In particular, it cannot capture state changes, or non-monotonically increasing resources such as stack size. As an alternative, we considered representing resource aware languages as an object language implemented in Ivor [10]. However, while it is possible to implement programs in this way, we found the notation too cumbersome to implement realistic programs.

The need to represent changing state while capturing the necessary resource properties led to an investigation of Domain Specific Embedded Language implementation, via parametrised monads [1]. We have found parametrised monads to be an effective implementation technique:

- In [5] we introduce the technique by implementing a file handling system which guarantees security properties — no file will be opened by an unauthorised user, no file will be accessed while closed, all files will be closed before exit.
- In [8], we show how the same technique can be used to verify properties of concurrent systems — resources must be locked before being read or written, and must be locked in an order which prevents deadlock.
- In [9] we will show the generality of the approach by applying it to a variety of domains, including memory management, which is directly applicable to embedded systems.

In each of these systems, we describe the language in terms of the effect each operation has on the resources on input and output. Operations may require a proof of a predicate, e.g. in our concurrency DSL (described in [8]):

\[
\text{LOCK} : \text{( \text{locked} : \text{ElemIs} i (\text{RState} k ty) \ tins) } \rightarrow \text{ (\text{priOK} : \text{PriOK} i \ tins) } \rightarrow \\
\text{Lang tins} (\text{update} i (\text{RState} (s k ty) \ tins)) \ \text{TyUnit}
\]

This describes locking a resource; before locking, a resource must have been locked \(k\) times (\(vk\) may be zero), there must be a proof that the priority ordering is correct with respect to the other resources (\(\text{PriOK} i \ tins\)), and the result is that the lock state has been updated so that the resource has been locked \(k + 1\) times. To use this operation in a program, we simply give the resource to be locked, and rely on the typechecker to generate the proof obligations (i.e. the predicates to be proved).

We have found that in this way, important resource properties with direct application to embedded systems (especially concurrency and memory management) can be captured with dependent types. Generative
programming techniques are an important aspect of our approach — especially, domain specific language implementation and multi-stage programming.

5 Academic Collaboration

As a result of work on this project, Edwin Brady has joined IFIP Working Group 2.11 on Program Generation, giving St Andrews the greatest concentration of researchers in the working group. His work on the design and implementation of Idris and associated examples ‘has led to links with Stephanie Weirich and the programming language group at the University of Pennsylvania, and he has been invited to join the committee to design a new dependently typed language, Trellys, in collaboration with Dr Weirich, as well as Tim Sheard of Portland State University and Aaron Stump of the University of Iowa. We also collaborated with Walid Taha’s group at Rice University, Texas, as planned, resulting in a valuable exchange of knowledge, including the implementation of staging annotations in Idris and dependent types in Concoqtion [15].

Edwin Brady has also continued his collaboration with Conor McBride of Nottingham University (now at Strathclyde), making two visits to Nottingham. They are continuing research into dependently typed language implementation in the context of Epigram [19].

Finally, the project has had close ties with other projects at St Andrews, including the EU FP6 project EmBounded (IST-510255) and SEAS DTC SEN-002, and led to connections with researchers in other fields at St Andrews and elsewhere.

6 Follow-on Projects

As a result of research in this project, several further potential projects have arisen. We have obtained a £1.1M EPSRC project to investigate the application of resource modelling to FPGA devices as part of MIMO communication devices (with Heriot-Watt, Edinburgh and Queens’ Belfast), obtained a 1-year support fellowship from the Royal Society of Edinburgh to investigate our ideas further, and obtained a £240,000 follow-on project as part of the MOD-funded SEAS Defence Technology Consortium (with Waterfall Technologies). We are also in the process of proposing further projects based on the idea of implementing verified domain specific language by embedding in a dependently typed host language in the context of stream-processing and network communication.

7 Publications

As shown on the form and detailed in this report, we have published several research papers during the course of the project. A further paper has been submitted, another is in preparation, and we anticipate that we will publish additional papers based on unpublished material in the coming months.

8 Conclusions

This has been a successful project. We have dealt with some very difficult technical problems and made significant progress towards solving some challenging problems in program verification. We are in a position to apply our results to a variety of fields, with their own difficult verification problems, such as parallel stream processing and network protocol verification. The project has furthered the career of Edwin Brady, who now has an academic position in St Andrews, in several ways. He is a member of IFIP WG2.11, will join the Trellys language committee, and has developed and strengthened existing links with researchers around the UK and internationally.

References


