Resource-Aware Programming*
Invited Paper

Walid Taha

Rice University, Houston, TX, USA
taha@rice.edu

Abstract. Traditional wisdom in programming language design suggests that there is a trade-off between expressive power and static guarantees. We describe a novel schema for designing a class of languages that we call Resource-aware Programming (RAP) languages. By taking into account the natural distinction between the development platform and the deployment platform for embedded software, RAP languages can alleviate the need for drastic trade-offs between expressive power and static guarantees. We describe our preliminary experience designing and programming in a RAP language for hardware design, and give a brief overview of directions for future work.

1 Introduction

Designers of embedded and real-time software must attend not only to functional specifications, but also to a wider range of concerns, including resource consumption and integration with the physical world. In current practice, the dominant medium for programming is various dialects of C. This is a puzzling state of affairs, given that: First, C is now over thirty years old, has many well-known limitations, including several well-known safety problems, and has a limited set of abstraction mechanisms; second, since then the programming languages community has produced new languages that address many of these safety problems, and developed several powerful abstraction mechanisms. Today, there is pressing need for addressing this issue. In particular, as new embedded hardware platforms continue to flow into the embedded systems market, the need for effective techniques for producing reliable software in a cost-effective manner becomes more pressing.

Real, technical challenges have hampered the adoption of language innovations in the embedded software domain. Possibly the most important reason is that it often appears as if a choice has to be made between expressive power and static guarantees. Expressive programming languages, often referred to as high-level languages, generally offer powerful abstraction mechanisms like higher-order functions, managed dynamic data structures, and general recursion. At the

* Supported by NSF ITR-0113569 "Putting Multi-stage Annotations to Work" and Texas ATP 003604-0032-2003 "Advanced Languages Techniques for Device Drivers."

same time, these languages provide little or no guarantees about resource utilization. Because such languages tend to deprive the programmer of control over resources that she *must* take full responsibility for, they are generally not well-suited for building embedded applications. Resource-bounded languages, such as state charts or synchronous languages, provide strong guarantees about the runtime behavior of programs. Because such languages generally deprive the programmer of constructs that allow her to write concise, structured, modular, and reusable programs, an unsafe language that provides more expressive power can be significantly more attractive in practice.

While there are several important innovations as well as ongoing efforts in the programming languages community to offer better trade-offs between these poles, an important insight has long been overlooked. Our key observation is that this apparent need to choose between expressive power and static guarantees can be often be avoided. *Resource-aware Programming (RAP) languages* are a class of languages aimed at addressing the problems described above by:

1. Providing a highly expressive *untyped substrate* supporting state-of-the-art abstraction mechanisms such as dynamic data-structures, modules, objects, and higher-order functions. The role of this substrate is to provide a common, unified model of the semantics of the whole computation, starting from what happens on the platform used to design the software, and extended to what must take place on the embedded platform where the software must ultimately operate. Because of their simpler reasoning principles and the wealth of results on statically checking them, our studies tend to use functional programming languages as untyped substrate [5]. In principle, these ideas should be applicable to any programming language.
2. Allowing the programmer to express the *stage distinction* between computation on the development platform and computation on the deployment platform. Expressing the stage distinction is, in principle, achieved by any language that can support program generation or that has a macro-expansion facility. But mechanisms based on strings or s-expressions would be insufficient, as they would interfere with the possibility of automatic static checking of programs *before* they are generated.
3. Using *static checking* to ensure that computations intended for execution on resource-bounded platforms are indeed resource-bounded. In fact, the ability to perform this kind of static checking is the most novel feature of RAP languages. To get an appreciation for the importance and the challenge involved in doing this, consider the analogous situation in the context of C: It would correspond to statically checking the safety and resource usage of C programs *before* they are pre-processed using configuration parameters for various target platforms.

The combination of these three ingredients allows the programmer to use sophisticated abstraction mechanisms in programs that are statically guaranteed to generate only resource-bounded programs. We expect that languages with these features can provide a solid bridge between traditional software engineer-

ing techniques on one side, and the specific demands of the embedded software domain on the other.

For general-purpose programming, the idea of statically checked generators has been studied extensively, largely in the context of *multi-stage programming* [3]. For general-purpose software, statically checked generators provide a mechanism for avoiding the runtime overhead typically associated with abstraction mechanisms such as functions and objects. For embedded software, the primary role of such generators will be to allow powerful abstraction mechanism to coexist with statically checkable properties on resource usage.

To date, our preliminary efforts to explore the idea of RAP languages have consisted of two main efforts: First, we have shown how a heap-bounded programming language can be extended with higher-order features [4]. Our experience in this study suggests that the static checking problems that arise in designing a RAP language can be non-trivial but nevertheless tractable. Second, we have shown how to use a two-stage language to concisely express Cooley and Tukey's recurrence that defines the Fast Fourier Transform (FFT) [1, 2]. These definitions are essentially program generators which can be used to generate exactly the butterfly circuit for FFT for any size 2^n. Our experience with this effort is discussed in the following section.

2 A RAP Hardware Description Language

RAP languages can play an important role in hardware design because, except for very high-end applications, verifying the correctness of hardware systems can be prohibitively expensive. In contrast, software languages are primarily concerned with issues of expressive power, safety, clarity, and maintainability. Software languages can provide abstraction mechanisms, which make designs more maintainable and reusable. They can also keep programs close to the mathematical definitions of the algorithms they implement, which helps with ensuring correctness. Hardware description languages such as VHDL and Verilog provide only limited support for such abstraction mechanisms. A RAP language for hardware circuits would allow us to capture the schema (or generator) for a family of circuits in an executable form. With such a schema, rather than having to verify circuits on a case-by-case basis, a unified substrate for the full process would enable the verification of a whole *family* of circuits en bloc.

A basic method for building a circuit schema in a RAP language has been proposed [1]. In addition to allowing us to implement a schema for FFT circuits concisely, following this systematic approach also yielded new insights into the relation between the FFTW and Split-radix implementations [2]. In this method, we start with naively-generated circuits that are correct by construction. In the case of FFT, this becomes evident because the schema is almost a literal transliteration of a textbook definition of the recurrence defining FFT. Then, more efficient circuits are correct as long as they are produced by systematic, verified improvements on a correct but naive generator. Note that these improvements can be carried out by improvements on the schema. Note also

that correctness is *not* achieved by verifying a naive generator and verifying a posteriori (post-generation) optimizations that fix up the result of the generator. This means that we replace the problem of verifying transformations to one of verifying modifications to one program: the generator.

2.1 Manifest Interfaces, Composition and Static Checking

As noted briefly in the introduction, statically checking generators can be hard to achieve using traditional type systems. For example, if strings, algebraic datatypes, parse trees, or even graphs are used to represent the generated program, they would only allow us to express a manifest interface with a type such as: `gen_fft : int -> circuit`, where `circuit` is the type we choose to represent circuits with. The static type `int -> circuit` says that `gen_fft` is a function that can only take an integer and can only produce a circuit. As soon as we start *composing* generators — for example, if we want to build a circuit that computes the FFT, performs a multiplication, and then computes the inverse FFT — we run into a problem: The type `circuit` does not provide any static information or guarantees about the consistency or well-formedness of the composite circuit. This is an instance of a general need for *manifest interfaces* that would provide us with enough static information to allow us to guarantee some degree of well-formedness on the result of the composite program. To illustrate, assume we are given two trivial generators which take no inputs and produce an AND-gate and an inverter:

```
and : circuit
inv : circuit
```

A meaningless composition arises if we write `let bad = inv --> and`, where the connect operator `-->` is an infix operator that has the type

```
circuit $\times$ circuit -> circuit
```

and which wires the output of its first circuit to the input of the second circuit. The problem is that the second circuit does not have just one input but *two*, and the type system does not prevent this error: All circuits just have type `circuit`.

It is generally desirable that the `circuit` type be as expressive as possible, but at the same time only express values that are circuit-realizable. For example, the programmer might want to use abstractions such as lists (or any other dynamic data structure) in describing the circuit, but will need to know as early as possible in the development process that these uses can be realized using finite memory [4].

2.2 Better Static Checking

Rather than using one concrete type to represent circuits, a RAP language provides an abstract datatype parameterized by information about the generated circuit. The type of the two trivial generators above would be:

```
and : (bool × bool -> bool) circuit
inv : (bool -> bool) circuit
```

The type of the connect operator `-->` would be refined from being

```
circuit × circuit -> circuit
```

to being

```
(α -> β) circuit × (β -> γ) circuit -> (α -> γ) circuit
```

where α, β, and γ are generic type variables that must always be instantiated consistently. With this extra information, the type system can reject the above `bad` declaration, because the type variable β cannot be instantiated to both the output of `inv` (which is `bool`) and the input of `and` (which is `bool`×`bool`). Note that the type of this function is similar to the type of the standard mathematical function composition operation of type `(α -> β) × (β -> γ) -> (α -> γ)`.

2.3 Safe Implementations of Domain-specific Optimization

To ensure that generated programs are well-typed and resource-bounded *before* they are generated, the circuit type constructor in a RAP language must remain abstract, meaning, that there is no mechanism within the language to allow the programmer to de-construct code once it has been generated. Providing constructs for traversing values of this type jeopardizes the soundness and decidability of static typing, and complicates reasoning about the correctness of programs written in these languages. At the same time, not being able to look inside the generated circuit descriptions means that a posteriori optimizations cannot be expressed within the language. While such optimizations can still be implemented as stand-alone source-to-source transformations *outside* the language, doing so invalidates the safety and resource-boundedness guarantees.

We distinguish two forms of a posteriori optimizations: Generic ones that are independent of the application, and ones that are specific to the application. Generic optimizations are generally well-tested and are less likely to invalidate the guarantees provided by the RAP setting. Such optimizations can be provided as special extensions of the language as long as they have been proven to preserve all guarantees. But domain-specific optimizations written by the programmer for a particular application are less likely to have been tested as extensively, and are therefore more problematic. At the same time, systems such as FFTW make a strong case for the practical importance of such domain-specific optimizations.

We were able to show that *abstract interpretation* on program generators can be used to avoid the need for a posteriori optimization [1]. This allows us to generate the desired circuits without losing the guarantees provided by RAP languages. The benefits of the proposed technique extend to the untyped setting, as it avoids the generation of large circuits in the first place, thus reducing the overall runtime needed to generate acceptable circuits. From the verification point of view, this approach replaces the problem of verifying a source-to-source transformation to that of verifying the correctness of a finite set of optimizations on *one* specific program: the generator.

3 Key Directions for RAP Research

The design space for RAP languages is huge, primarily because there are numerous notions of resource-boundedness and languages that can be considered for the deployment platform, as well as the numerous abstraction mechanisms that may be desirable on the development platform. A systematic survey is therefore beyond the scope of this paper. However, there are a number of broad directions that we expect to be important to progress in this area:

- Extensions of traditional static analysis techniques (including type systems) to work in a generative setting. We expect our own efforts to focus on analysis that have direct applications in challenging domains, such as device drivers, control systems, and hardware description languages.
- Better understanding of the process of writing RAP programs, including further study of the use of program structuring mechanisms such as monads, as well as the use of abstract interpretation as a programming technique for implementing domain specific optimizations.
- Support for certification. In particular, while previous work on RAP languages have so far focused on static guarantees, the execution model on the development platform can be naturally extended to preserve the proof behind this guarantee, and this proof can then be produced along side the deployment platform computation. Such a certificate can be verified independently of the generation process, much in the same way as proof-carrying-code is used to verify the safety of software received from an untrusted source.

Acknowledgment: Anthony Castanares, Emir Pašalić and Kedar Swadi have kindly read and commented on drafts of this paper.

References

1. Oleg Kiselyov, Kedar Swadi, and Walid Taha. A methodology for generating verified combinatorial circuits. In *the International Workshop on Embedded Software (EMSOFT '04)*, Lecture Notes in Computer Science, Pisa, Italy, 2004. ACM.
2. Oleg Kiselyov and Walid Taha. Relating FFTW and Split-Radix. In *Proceedings of the International Conference on Embedded Software and Systems*, 2004. Appears in this volume.
3. Walid Taha. A gentle introduction to multi-stage programming. In Don Batory, Charles Consel, Christian Lengauer, and Martin Odersky, editors, *Domain-specific Program Generation*, LNCS. 2004.
4. Walid Taha, Stephan Ellner, and Hongwei Xi. Generating Imperative, Heap-Bounded Programs in a Functional Setting. In *Proceedings of the Third International Conference on Embedded Software*, Philadelphia, PA, October 2003.
5. Walid Taha, Paul Hudak, and Zhanyong Wan. Directions in functional programming for real(-time) applications. In *the International Workshop on Embedded Software (EMSOFT '01)*, volume 2221 of *Lecture Notes in Computer Science*, pages 185–203, Lake Tahoe, 2001. Springer-Verlag.