Automatic Parallelization and Transparent Fault Tolerance

Kei Davis[1], Dean Prichard[1], David Ringo[1,2], Loren Anderson[1,3], and Jacob Marks[1,4]

[1] Los Alamos National Laboratory, Los Alamos, NM, USA
kei.davis@lanl.gov,
WWW home page: http://ccsweb.lanl.gov/~kei/
[2] University of New Mexico, Albuquerque, NM, USA
[3] North Dakota State University, Fargo, North Dakota, USA
[4] New Mexico Institute of Mining and Technology, Socorro, New Mexico, USA

Abstract. A claimed trend is an increasing appreciation and practice of functional programming in scientific computing, and in particular strict (by default) functional programming, in various guises. Besides making large programs easier to reason about, this enables automatic parallelization to various degrees. A second claimed trend in both scientific computing and functional programming is the development of mechanisms for transparent fault tolerance. Our project goal is the demonstration of a light-weight, higher-order, polymorphic, pure functional language implementation in which we can experiment with automatic parallelization strategies, varying degrees of strictness, and mechanisms for transparent fault tolerance. We do not consider speculative evaluation, or semantic strictness inferred by program analysis, so potential parallelism is dictated by the specified degree of strictness.

Our development has been greatly informed by the extant literature and we indicate both where we follow, and deviate from, these sources.

Keywords: Strict pure functional programming, automatic parallelization, transparent fault tolerance, Haskell, STG, GHC, GHC Core.

1 Background and Motivation

The practice of so-called high-performance scientific computing (HPC) is, overall, highly conservative with respect to change. Even as the USA and other governments commit to a push for exascale computing (10^{18} FLOPS in a single system) within the next decade [22,1,4] a respected member of the HPC community recently stated publicly that "Fortran is essential for exascale programming" [25]. Existing US government laboratory HPC codes run to hundreds of thousands of lines of Fortran each and complete rewrites are infeasible.[1] C++ has become more common for new code starts, but their nature remains the same: huge codes

[1] The codes of our lab's British counterpart are similar.

with fair to non-existent encapsulation of side effects, with multiple levels of parallelism sometimes poorly abstracted, especially at the thread (shared-memory) and computational accelerator (e.g. GPU) levels. Thread-level parallel efficiency can be good when the parallel model is bulk synchronous; tasking models are generally avoided unless abstracted by, e.g., a C++ template or runtime library, because of the programming complexity.

1.1 A trend: strict, pure functional programming in HPC and FP

Interest in functional programming for HPC in general is not new as evidenced by the Workshop on Functional High Performance Computing, held in conjunction with the International Conference on Functional Programming, now in its fifth consecutive year. A more precise characterization of the trend is that there is a small but growing understanding by practicing computational scientists that *pure functional semantics*, in some form, is essential to reining in the complexity of ever-evolving scientific codes and informing the designs of new ones, and that strict semantics tend to be preferred. Such semantics are expressed in a number of ways as highlighted following.

Rely on programmer discipline. Legacy Fortran scientific codes likely represent worst practices in state encapsulation, with most data global and accessible to all; indeed, Fortran was designed to make this easy via `COMMON` blocks—aggregates of global variables—which further obfuscate program meaning by allowing arbitrary naming of a block's variables at a subroutine/function granularity, and requiring that entire blocks be brought into scope. It is almost ironic, then, that of the mainstream high-performance scientific programming languages (Fortran, C, and C++), Fortran 95 was the first to introduce the `pure` function qualifier, and with enforcement by the compiler. Otherwise it is entirely up to the programmer to make functions composable and thread safe.

Obey constraints prescribed by a parallel runtime system. There are a number of parallel runtime systems that encourage (but cannot enforce) a pure functional style. We highlight Stanford's Legion runtime as an example [13].

Using Legion, all non-function-local variables, or *regions*, must be requested from the runtime system. Regions may be shared among *tasks* (essentially C++ functions), and each task is prescribed access privileges (read, write, read/write, etc.) and coherency requirements for each region to which it will have access. Other than registered access to regions, tasks must not access non-constant global data. Thus the inputs and outputs of tasks are exactly known to the runtime system. In serial execution these access requirements are notionally superfluous, but in parallel execution the runtime can dynamically calculate the data dependency graph and relax the programmatically defined serial order of

task execution to a partial, and therefore parallelizable, order. Very high parallel efficiency has been demonstrated on extremely large computing systems [14].[2]

Use a pure functional language. Twenty years ago the US Department of Energy laboratory complex was presented with the most highly parallel (and thereby performant) implementation of a pure functional language to date, to no effect whatsoever [21].[3] Today, however, there is a small number of laboratory HPC practitioners who are developing an appreciation for pure functional programming in Haskell. However, they see their needs as different than other Haskell users: performance is paramount, array/vector is a (perhaps *the*) primary data structure, and non-strictness or laziness is more of a nuisance—even a serious hindrance—than a convenience, much less ever algorithmically essential in the sense of Bird's *repmin* [16], Johnsson's general attribute grammars [27], or other instances of *tying the knot* [9].

A common complaint among these aspiring high-performance functional programmers is the need to get strictness just right: to remove space leaks, achieve competitive performance, get I/O and inter-process communication to work as they expect, and so on. More specifically, they expect to be able to reason about space and time usage, and order of evaluation, using their existing mental models. While they are happy to write code that is strongly reminiscent of the numerical examples in Hughes' classic piece (whether they've read it or not) [26], they would also be happy to write in a slightly modified style that does not require non-strictness and so that their programs behave as they expect.

Much anecdotal evidence suggests that many others have similar difficulties with semantics that are lazy (or non-strict) by default, with the most telling recent concrete evidence being the push for the `Strict` and `StrictData` language extensions newly available in GHC 8.0. Quoting the documentation:

> High-performance Haskell code (e.g. numeric code) can sometimes be littered with bang patterns, making it harder to read. The reason is that laziness isn't the right default in this particular code, but the programmer has no way to say that except by repeatedly adding bang patterns [7].

It is a fact that *HPC practitioners are generally not computer scientists*, they are domain scientists (physicists, chemists, etc.) or computational scientists specialized in numerical or numerically-oriented algorithmic methods (e.g., Lagrangian, Eulerian, adaptive mesh refinement). The concept of a space leak, for example, caused by other than a missing explicit deallocation statement (C `free`, Fortran `DEALLOCATE`, etc.) is difficult to understand and reason about.

More generally, *task-based* parallelism is becoming increasingly popular for HPC, often in conjunction with a higher-level (typically inter-node) bulk synchronous model usually implemented with the Message Passing Interface (MPI) [34].

[2] We note that the PI of the Legion project was previously a co-developer of a strict, arguably pure functional language implementation [10,12], in collaboration with John Backus as a refinement of his FP language [11].

[3] In 1996, on the LANL Thinking Machines CM-5, the Top 1 supercomputer in 1993 [8].

In the imperative world, example systems include the aforementioned Legion, Intel's Threading Building Blocks [41], and the HPX C++11/14 extension [49], all of which also seek to make parallelism automatic and transparent. While the concept of task is somewhat vague, the general idea is a unit of computation with no or minimal/constrained side effects. A function in a pure functional language could be regarded as the essence of the concept of a task. All of these systems are of course in languages with strict function-call semantics.

1.2 Another trend: transparent fault tolerance in HPC and FP

In the quest for computing platforms approaching exascale, it is widely recognized that new mechanisms for fault tolerance will need to be built into the software stack. Current practice in HPC is checkpoint/restart, wherein at various points in time sufficient global program state is dumped to backing store such that the program, in case of a fault, can be (manually) restarted from the last stored state. Most implementations are explicit, though there has been some success with transparent (invisible to the programmer) systems. Manually wiring checkpoint/restart into a large-scale application is not only burdensome and error-prone, it can severely warp the engineering of the application because it must be designed such that consistent, essential state can be readily captured. Finally, the scalability of global checkpoint/restart is reaching its end as data transfer to backing store becomes an increasing bottleneck [20,5].

In the imperative world side-effect-free tasks could be safely restarted after failure, needing only their well-defined inputs and not arbitrary access to mutable global state. The Legion system is being augmented with this capability, and in such a way that it will be largely transparent to the programmer. Ericsson's quasi-functional language Erlang and its runtime system were designed for fault tolerance [18]. It has long been recognized that the side-effect-free (or revocably side-effecting) parts of a functional program could be safely replicated or restarted. The recent work of Stewart impressively demonstrates this, in a pure functional setting, on a large-scale distributed-memory system [47].

We claim, then, that there is a trend, in HPC at least, towards *strict-by-default pure functional programming* and *automatic parallelism*. Second, we claim that there is a trend in high-performance computing, and functional programming, towards *transparent fault tolerance*.

2 Project and Goals

Given strict functional semantics wherein function arguments may be safely evaluated before function evaluation or call, it is safe to evaluate the arguments, and the function, in parallel, and that this applies recursively in the (dynamic) expression tree (or graph), and similarly for data constructors as functions. Our project is the light-weight implementation of a pure, higher-order, polymorphic, functional language and runtime system with which we can experiment with automatic parallelization strategies with varying degrees of language strictness,

and secondarily, with mechanisms for transparent fault tolerance. Light-weight has several implications. First and foremost it means that *we are not attempting to compete with GHC in any respect*—succumbing to the temptation of recreating a significant subset of GHC language, type system, or runtime features, or significant code optimization, would simply divert us from our central goals.[4] For another, it means that we are not undertaking significant morphing of GHC itself: because this is in large part an undergraduate student project it must be kept as simple as reasonably possible. Our initial requirements are that

- The implementation must be feasible in terms of available effort;
- The language must be strongly typed, implicitly or explicitly. More specifically, the language should explicitly support both Hindley-Milner polymorphism and unboxed types [38];
- Straightforward direct linkage with C, preferably without a sophisticated foreign function interface, is essential;
- The language should be *currying-friendly* as described by Marlow and Peyton Jones [32];
- Choice of degree of strictness should be easily selected;
- Sharing, in the sense of what *lazy* means vs. merely non-strict, should be preserved for all degrees of strictness;
- The generated code is C;
- A certain consistency in tail calling in C must be achieved.

For feasibility we restrict ourselves to implicit parallelism on shared-memory multiprocessors. The requirement for tail calling is not primarily for efficiency, rather for an efficient parallel implementation. Achieving this, and thereby sufficiently limiting the use of the native C stack, will be discussed at some length.

A minor, informal goal is to gain a sense of how often relatively inexperienced functional programmers make essential use of non-strictness.

3 Design and Implementation

Our wish list has much in common with the GHC Haskell implementation, and indeed GHC is part of our master plan. In brief, the phases of the GHC compiler are (1) parsing and type checking; (2) transformation to the GHC *Core* intermediate representation [48,50]; (3) analysis and transformation on Core; (4) transformation of Core to the *STG* intermediate representation [37]; and (5) transformation of STG to machine code via various imperative representations (Cmm, LLVM, C, native code generation).

In short, our plan is to use GHC as our front-end to generate one of its intermediate forms, STG or Core, then escape to our system from there. There are various possible ways to accomplish the this: using GHC's option to dump STG or Core as text, hacking GHC itself, and, possibly, using GHC as a library [2]. Initially we started with our own STG.

[4] We estimate that we will have, to within an order of magnitude, one person-hour of total available development time for each person-year that GHC currently represents, or in other units, one second for each hour.

3.1 STG Language

Our incarnation of STG, as shown in Figure 1, is meant to evoke both an abstract and concrete syntax. This formulation and presentation is adapted from Marlow and Peyton Jones' *Eval/Apply* paper [32], to which we will subsequently refer as *EA*. A *BLACKHOLE* object, signifying a *THUNK* being evaluated, is only a run-time entity and not part of the language. Omitted is the syntax for algebraic data type definitions which is given in Appendix B.

Variable	$f,\ x$			Initial lower-case letter
Constructor	C			Initial upper-case letter
Literal	lit	::=	$i \mid d$	Integral or floating point literal
Atom	a	::=	$lit \mid x$	
Expression	e	::=	a	Atom
		|	$f\ a_1 \ldots a_n$	Application, $n \geq 1$
		|	$\oplus\ a_1 \ldots a_n$	Saturated primitive operation, $n \geq 1$
		|	`let` { $odecl_1$; ... ; $odecl_n$ } `in` e	Recursive let, $n \geq 1$
		|	`case` e `of` x { alt_1 ; ... ; alt_n }	Case expression, $n \geq 1$
Alternatives	alt	::=	$C\ x_1 \ldots x_n$ `->` e	Pattern match, $n \geq 0$
		|	_ `->` e	Default
Heap objects	obj	::=	`FUN` $x_1 \ldots x_n$ `->` e	Function definition, arity $= n \geq 1$
		|	`CON` $C\ a_1 \ldots a_n$	Saturated constructor, $n \geq 0$
		|	`THUNK` e	Thunk—explicit deferred evaluation
		|	`PAP` $f\ a_1 \ldots a_n$	Partial application
		|	$BLACKHOLE$	Evaluation-time black hole
Object decl.	$odecl$	::=	$x = obj$	Simple binding
Program	$prog$	::=	$odecl_1$; ... ; $odecl_n$	Object and data defns, distinguished `main`

Fig. 1. STG syntax

With a few exceptions the semantics of STG is what a Haskell programmer familiar with unboxed types would expect for a higher-order, polymorphic, pure functional language. Points to note are, first, that `case` is strict in the scrutinee (the e in `case` e `of`), so `case` is similar to `pseq` (as opposed to `seq`) in Haskell,

that is, the scrutinee is evaluated first. Second, constructors are not functions, rather they are named in the construction (heap allocation) of a data object. More generally, the operational notion of allocating a heap object is explicit: this is the operational semantics of `let` bindings. Numeric literals are considered constructors in pattern matching. Finally, deferred evaluation is explicit: a deferred expression evaluation is encoded as a *THUNK*.

3.2 Degrees of strictness

First some words about terminology. For all degrees of strictness, sharing (as implied by lazy but not by non-strict) of heap-allocated objects is preserved so we will not use the term lazy and simply refer to strictness properties.[5] As is common we will use strict in an operational sense, meaning that argument(s) are evaluated before being passed in a function call, rather than the denotational sense, unless stated otherwise. Similarly, when referring to language semantics we refer by default to the operational rather than denotational semantics.

Greater strictness implies more opportunities for automatic parallelism, possible losses or gains in space and time efficiency, and less expressiveness. We seek to explore the interplay of all of these in various ways.

GHC 8.0 gives three defaults: conventional non-strict, constructor strict, and function- and constructor strict, but there are possible variations. The obvious variation is function, but not constructor, strictness. In Haskell the distinction between constructors and functions is somewhat blurred (outside of data definitions and pattern matching) but in STG the distinction is clear.

Other possible variations concern the notional arity of the underlying function being applied. This is readily explained in terms of the operation of the STG abstract machine. Our partial description is based on that given in *EA*.

In STG the arity of a function is defined as the number of formal parameters in the function definition, thus `id x = x` has by definition arity one regardless of the type of the argument it might be applied to (e.g., some function type). We distinguish *application*, e.g., `f x y`, which is a syntactic notion, from actual function *call*. Thus `id id x` is an application of `id` to two arguments, but the underlying *FUN* (user- or system- defined function) when called is given exactly one argument (the first one).

In the eval-apply model, given application `f x y`, in the general case, with standard (non-strict) semantics, `f` is first evaluated and the arity of the underlying *FUN* or *PAP* (partial application of a *FUN*) is determined—in the case of *PAP* the arity is that of the underlying *FUN* less the number of arguments previously provided. If the arity is greater than two a new *PAP* is created with arguments `x` and `y` inserted and the new *PAP* returned. If the arity is equal to two the underlying *FUN* is tail-called (jumped to) with any previous arguments (in case of *PAP*) and new arguments `x` and `y`. If the arity is less than two (it

[5] We can readily simulate the absence of sharing (i.e, call by name instead of call by need), in the serial case at least, simply by not updating *THUNK*s after evaluation, resulting in, for example, exponential running time for simple mergesort on lists.

must be at least one, so one in this case) the underlying *FUN* is called with any previous arguments (in case of *PAP*) and new argument `x`, then the object returned applied to remaining argument `y`.

Given application `f x y`, when might `x` and `y` be evaluated? In the non-strict semantics evaluation is on demand, after `f` has been evaluated, the underlying *FUN* actually called (i.e., not just partially applied and wrapped up in a *PAP*), and one or both of those arguments possibly forced in subsequent evaluation, assuming that they weren't forced in the evaluation of `f` itself.

In a more strict semantics one could devise various evaluation orders, but as far as distinguishing termination properties it seems that these come down to two reasonable possibilities, namely whether on construction of a *PAP* the (new) arguments are forced or not. In other words, the question is whether application is strict, or function call is strict. These are in fact distinguishable because (in Haskell, for example) the first argument to `seq` could evaluate to a *PAP*.

This then gives three reasonable operational modes which are strictly ordered in terms of termination properties: non-strict, function strict, and application strict. The other dimension we vary is whether or not constructors are strict. Other dimensions are possible, for example whether `let` bindings are strict, but we restrict ourselves to function/application and constructor strictness. The STG machinery makes the choice of evaluation strategy easy to change, though changing STG generation can be more efficient by avoiding the unnecessary creation of *THUNK*s as discussed in Appendix A.

3.3 STG to C

The back-end is written in Haskell and generates C code; the runtime system is written in C/C++. This gives us the extensive optimization performed by modern C compilers such as GCC and Clang/LLVM. The system fairly faithfully implements the STG machine as described in *EA* and dynamically illustrated by Pope's Ministg [15] with the exception of the state transition rules which may be different than those for non-strict semantics. Sharing is preserved.

Proper tail calls in C. Parallelism is achieved by multi-threading; our model is based on Harris et al.'s. [23]; later developments by Marlow et al. are beyond our aspirations [31,30]. To achieve efficient parallel execution they identified the need for the ability to examine and unwind the stack in quashing concurrent evaluation of the same *THUNK*. This requires that use of the C stack be highly constrained if we are to avoid complex and ugly hacks using, e.g., *setjmp/longjmp*, or non-portable code to directly manipulate the C stack, complicating an already non-trivial implementation effort. We therefore must limit the use of the C stack to calls to the runtime, in other words, when executing code generated from the user program (and not in a runtime call), there must be no growing of (pushing return addresses or data onto) the C stack. To clearly distinguish what a C call (a syntactic concept) can compile to, we will refer to a machine-level call—pushing a return address onto the native C stack—as *call-with-return*, and to a machine-level jump as *jump*.

First we establish that this is possible in principle on x86-64/Linux. By maintaining our own data and control (continuation) stack all generated C functions can take a uniform (including empty) set of explicit arguments (passed in registers) and have uniform return type. If we arrange that generated C code has only calls (other than runtime or other external calls, which do not call back to user-program code) in tail call positions, that is, immediately before a return in the control flow graph, then at non-zero optimization levels both GCC and Clang/LLVM will compile C calls to jumps instead of calls-with-return. When using GCC we can also use an explicit set of machine registers for passing/returning values, much like GHC using the LLVM back-end with its custom GHC calling convention.

The STG abstract machine is implemented using a combined data and continuation stack. Continuations contain a code pointer, optionally some data, and layout information for the benefit of the garbage collector. When the evaluation of a C function (corresponding to some STG code) is completed its return is by an indirect jump using the code pointer in the continuation on the top of the stack ("returning through the continuation stack").

EA describes a family of generic C-- *stgApply* runtime functions used for applications of non-known (not statically identifiable) functions, and applications of known functions to other than their arity number of arguments. There the object to be applied, if a *THUNK*, is first evaluated with call-with-return. When a (resulting) *FUN* or *PAP* is applied to more than its arity number of arguments, it is called-with-return with its arity number of arguments, and the resulting value applied to the remaining arguments. Call-with-return becomes even more convenient for strict evaluation: arguments can be evaluated in a tight loop, and similarly for strict constructors, if the STG generator (human or otherwise) has not arranged for them to be already evaluated.

These uses of call-with-return, which in general cannot be inlined, grow the C stack, and without bound. Our solution uses a single C *stgApply* function and an auxiliary function with no calls-with-return. This is straightforwardly realizable because the stack is explicit, and with modern C compiler call optimization the continuation style enables a jump mechanism where the *code labels*—entities that name blocks of code, can be treated as first class (e.g., stored for later use), and serve as targets of a jump—are C function addresses (c.f. §6.1 [37]).

3.4 Type system

Type information is required for code generation. For example, at the coarsest level it is necessary to know whether values are boxed, or otherwise represented by pointers, to avoid expensive tagging. Because we do not yet get type information directly from GHC we use a subset of Haskell syntax (e.g., no named fields) for defining algebraic data types, extended with built-in and user-defined unboxed types. We perform standard Hindley-Milner type inference and enforcement following Heren et al. [24], again extended with unboxed types. The extensions to the type inference algorithm, syntax, and restrictions for unboxed

types follow Peyton Jones and Launchbury [38]. For convenience we allow general recursive `lets` that are implicitly factored in the usual way to maximize let-polymorphism; this would normally already be performed by a front-end.

4 Current Status, Future Work

An almost unbounded amount of interesting development suggests itself; one such path could in principle lead to the recreation of GHC. However, given the minuscule budget (and therefore the employment of student interns) we must maintain sharp focus on the main goals: implementing and evaluating mechanisms for automatic parallelism and fault tolerance. Currently we have a fully functioning serial implementation including

- STG parser,
- mini-Haskell parser, mini-Haskell to STG transformer,
- type inferencer,
- various analysis and transformation passes,
- code generator,
- runtime system including a garbage collector,
- continuous integration and testing system,
- extensive test suite, automated testing.

The *mini-Haskell* front-end is detailed in Appendix B.

We use a continuous integration system augmented with logic to categorize programs in a large and growing test suite according to which degrees of non-strictness they require to terminate correctly, and to provide other logging and debugging information. This is detailed in Appendix C.

While we can parse and type-check more general unboxed types, code generation and runtime support is currently limited to built-in single-word (64-bit) or smaller types (e.g., `Int#`, `UInt#`, `Float#`, `Double#`) and user-defined enumerations (simple sums), e.g., `data unboxed Bool# = False# | True#`.

For simplicity the implementation is 64-bit only. The current, primitive parallel runtime is based on Pthreads and a non-blocking work queue [35]. We are currently developing a more realistic parallel runtime, experimenting with using the Argobots user-level thread library [42]. Bridging the gap between GHC STG and our STG, and the corresponding augmentation of the runtime is showing early signs of success. An instrumentation infrastructure, integrated with the testing framework to record and present time and space run-time statistics such as number of heap and stack allocations and maximum stack depth, is planned for the near future. The fault-tolerance mechanisms remain future work. Parallel garbage collection is likely out of scope.

5 Related Work

There is a vast body of related work; we highlight only the very most closely related.

Strict but still pure. The concept of strict but still pure in some sense dates back to the concept of applicative order reduction in the lambda calculus. In terms of a notionally concrete programming language the first thorough explications may be Backus' FP [11] and follow-on FL implementation [10,12]. A notable argument for the potential merits of a strict-by-default pure functional language over a non-strict one is by Sheard, who also provided a very simple embedded implementation [43]. A plug-in for GHC (prior to 8.0) exists that makes function bodies strict in their non-recursive `let` bindings using a transformation on GHC Core that recursively expands those `let` bindings into `case` bindings [17,6]. The Disciplined Disciple Compiler project appears to intend implementation of a strict, higher order, functional language (or family of languages) compiler and has been underway for some time, but the emphasis appears to be on a much more complex type system for handling stateful computation, and the prospects for its eventual completion are unknown [28].

Strict, pure, auto-parallelizing. NVIDIA's NOVA language, and corresponding compiler, implements a strict ("call by value"), pure, polymorphic, higher-order functional language with built-in parallel operators (e.g., *map*, *reduce*, and *scan*) [19]. Three back-ends were implemented: for sequential C, parallel (multi-threaded) C, and CUDA C, the lattermost targeting NVIDIA GPUs.

GHC as a front-end. The Intel Labs Haskell Research Compiler (HRC) is the most closely related work to ours of which we are aware [29,36]. As in our planned approach they use GHC as the front end, but exit GHC at the final GHC Core stage, just before transformation to STG. The essence of their effort is to target an existing, highly optimizing, Intel functional language compiler that is "largely language agnostic" and performs (among many other optimizations) SIMD vectorization.

Don Stewart described an implementation wherein GHC was used as a Haskell compiler front-end, dumping STG code as text, parsing that text, then translating to Java code for compilation and execution on the Java Virtual Machine [46].

Fault tolerance for functional programming. Pointon et al. developed a limited fault tolerance mechanism for Glasgow distributed Haskell based on distributed exception handling [40]. However, it appears that their scheme for handling failed processing elements correctly may have never been implemented [51].

More recently, Robert Stewart extended Haskell Distributed Parallel Haskell (HdpH) [3] to be fault resilient at an impressive scale—1400 cores on a distributed memory system. Here the key recovery strategy is task replication [47].

6 Acknowledgments

Funding for this project has been provided by the DOE NNSA LANL Laboratory Directed Research and Development program award 20150845ER; the National Science Foundation Science, Technology, Engineering, and Mathematics Talent Expansion Program (NSF STEP) program for undergraduate students;

and the Department of Energy Science Undergraduate Laboratory Internship (DOE SULI) program.

Los Alamos National Laboratory is managed and operated by Los Alamos National Security, LLC (LANS), under contract number DE-AC52-06NA25396 for the Department of Energy's National Nuclear Security Administration (NNSA).

A (Ab)Using GHC

The bridge to full Haskell is in progress, with GHC providing much of the structure. We are using GHC as a library (as opposed to modifying its source to use our code generation and runtime system). This choice was made primarily due to the size and complexity of a project as large as GHC. Fully understanding and modifying even its build system would be a significant undertaking.

We are using GHC's pipeline to translate Haskell source to GHC's representation of the STG language, which we then translate to our own. Up to the STG generation stage, this usage is equivalent to GHC's `--make` behavior, which allows us to benefit from the module dependency analysis, typechecking, Core-level transformations and informative error messages for invalid programs.

Though we follow the same rough idea as GHC in implementing the STG abstract machine, this bridging process is not entirely straightforward. The challenges we've seen fit into two categories.

A.1 Producing Full STG Programs

When compiling a standard Haskell program with GHC, modules imported from other packages have already been compiled, and GHC simply links the user object code with them. Our compiler has no facility for linking GHC-produced object code, so we need it to produce *full* STG programs built up from primitive operations. GHC does not offer object-to-STG translation, so this means we must produce our own Prelude and supporting Base libraries to be included with every program, and decoupling Haskell from GHC's wired-in types as much as possible.

This is partially accomplished with extensions like `NoImplicitPrelude` and `RebindableSyntax` (which implies the former). We cannot get a complete separation, however: Numeric literals are tied to GHC's `Integer` and `Rational` types, and guards are tied to GHC's `Bool`. Lists and tuples, being special syntactic forms are also tightly wired-in. To get around this, we provide placeholder definitions that parallel any that we cannot replace, and use them to crudely translate the exceptional cases.

A.2 Navigating GHC's Internals

One of the goals of using GHC as a library was to avoid being overwhelmed with the necessary complexity of such a large and mature tool, but this has not

been entirely unavoidable. Simply producing STG without incurring any (non-temporary) file creation proved not to be a difficult task, and can be accomplished using only the interface exposed via the primary API, the `GHC` module. This abstract STG is even ostensibly very close to our own, but the auxiliary data structures (e.g. those encapsulating identifier information and types) are rich and deeply nested. Determining what is strictly necessary for our backend and what we have no implementation of (e.g. the Foreign Function Interface) has taken considerable time and effort.

It is a testament to the talents of the GHC contributors and maintainers that we have managed to come even this far. Their documentation and exposed interfaces have made navigating and comprehending the project internals much more pleasant than expected.

B Mini-Haskell

As mentioned, a current near-term goal is to use GHC as our front-end to generate STG (or Core) as input to our system. In the meantime, as a stop-gap measure to make the generation of test cases easier, we have implemented a *mini-Haskell* front-end that supports a subset of standard Haskell syntax with *MagicHash* enabled, with a single extension. Recall that *MagicHash* allows `#` to be a suffix of an identifier, or denote an unboxed literal value. Our extension (as for STG) is the provision of the optional `unboxed` keyword in datatype definitions to allow user-defined unboxed data types, e.g.,

```
data unboxed Bool# = False# | True# .
```

Writing STG in the concrete syntax conceived for our project is tedious and prone to errors. Compared to Haskell, the eventual source language for the compiler, STG is relatively verbose, requiring explicit block and expression delimiters, and classification of heap objects. Though its general structure is similar to primitive Haskell, it lacks many of the conveniences that make Haskell a pleasure to program in. The motivation for a more expressive language for testing our compiler and runtime system in their nascent stages is then clear. This language, mini-Haskell, was progressively "grown" from the STG syntax towards a subset of Haskell itself, removing much of the frustrating syntactic cruft and adding some of the core features that are often taken for granted in Haskell (and other languages). An alternative approach would translate from mini-Haskell to GHC Core, then either to STG or, in the case that the back end eventually take, compiling Core directly.

An informal specification of the mini-Haskell grammar is given in Figure 2. There is significant overlap with STG resulting from the structural similarities, but the redundancies are included for completeness.

Because mini-Haskell was designed to supplement STG for test generation, it makes sense to talk about it in terms of the differences between the two and how mini-Haskell is translated into STG for the back-end. Here, the differences are given in order of perceived utility, or importance in transitioning from STG to mini-Haskell.

Variable	$f,\ x$			Initial lower-case letter
Constructor	C			Initial upper-case letter
Literal	lit	::=	$i \mid d \mid i\# \mid d\#$	Integral or floating point literal # suffix denotes unboxed value
Atom	a	::=	$lit \mid x \mid C$	Constructors are first class
Expression	e	::=	a	Atomic expression
		|	$e\ e$	Expression application
		|	$\backslash\ pat_i$ `->` e	Lambda expression, $i > 0$
		|	`case` e `of` alt_i	Case expression, $i > 0$
		|	`let` $odecl_i$ `in` e	Recursive let expression, $i > 0$
Pattern	pat	::=	$C\ pat_i$	Nested constructor matching
		|	lit	Match numeric literals
		|	x	Bind to variable
Alternative	alt	::=	pat_i `->` e	$i > 0$
Type	$type$	|	C	Concrete type (constructor naming)
		|	x	polymorphic type variable
		|	$type$ `->` $type$	Function type
Object decl.	$odecl$	::=	x `::` $type$	Type signature
		|	$x = e$	Simple binding
		|	$f\ pat_i = e$	Function declaration e cannot be an unboxed literal
Constructor defn.	con	::=	$C\ type_i$	$i \geq 0$
Datatype defn.	$ddecl$	::=	`data` [`unboxed`]	User-defined data type
			$C\ x_i =$	$i \geq 0, n > 0$
			$con_1 \mid \ldots \mid con_n$	
Program	$prog$	::=	$(o\mid d)decl_i$	Object and data defns

Fig. 2. mini-Haskell syntax

Removal of explicit heap objects. In STG, all object definitions require explicit classification with `FUN`, `PAP`, `CON`, `THUNK`. This classification can be inferred from context by the compiler so these keywords are not necessary. Functions can then be defined in Haskell style with the formal parameters following the function's identifier on the left side of the equality. With these simple changes the syntax becomes a subset of Haskell. Heap objects are still only created at the top level or in `let` expressions, so the translation is straightforward. `FUN` objects are identifiable by the inclusion of these parameters. A `PAP` object is created when a known function is partially applied. `CON` objects are created when constructors are fully applied. Every other object becomes a `THUNK`.

Avoiding building thunks. As *EA* points out, there is no need to build a thunk for the scrutinee of a `case` expression because if the `case` is evaluated the scrutinee is certain to be evaluated. Similarly, if a function is strict there is no need to build a thunk for each of its arguments (assuming boxed argument type). Because STG allows only *atoms* (variables or literal values) in argument positions, naive translation of Haskell `f e` to `let {x = e} in f x`, followed by naive code generation, will result in an unnecessary heap allocation. An alternative is to translate to `case e of x {_ -> f x}`, which we regard as a role of a front-end such as mini-Haskell.

The layout rule. Mini-Haskell implements Haskell's layout rule in an almost direct translation of the specification given in section 10.3 of the Haskell 2010 Report [44]. The notable exception is the omission of any attempt to define the `parse-error` function. This is apparently a tricky spot for Haskell compilers (even GHC has not always had it right) so, in the name of tradition, it remains a tricky spot for our mini-Haskell front end. The edge cases that it will incorrectly handle as a result of this are exactly that: edge cases. As such, they are easy to avoid in practice.

Expression to expression application. In STG function application must be to atomic values, i.e., either literals or variables. This often requires introducing bindings with either `let` or `case` expressions. Consider the `append` function that concatenates its two list parameters, which can be expressed in STG by

```
append = FUN(l1 l2 ->
            case l1 of
              { Nil -> l2;
                Cons hd tl -> let { rec = THUNK(append tl l2);
                                    result = CON(Cons hd rec) }
                              in result })
```

In mini-Haskell expressions may be applied to other expressions, so the recursive call to `append` can be parenthesized. There is no need for the `let` expression:

```
append l1 l2 = case l1 of
                 Nil -> l2
                 Cons hd tl -> Cons hd (append tl l2)
```

This also serves as a good example of how expression to expression application is transformed into STG. Any non-atomic expression in an application expression is bound to a heap object in a generated `let` expression. These expressions are thus properly "atomized" for STG. The above mini-Haskell code is transformed into STG that is structurally identical to the STG code that precedes it.

Fancy pattern matching. STG supports pattern matching but only in a simple form and only in `case` expressions. Mini-Haskell extends this to something approaching what is found in Haskell. Functions, including lambda expressions, may match on all their arguments. Named functions may have multiple definitions to exhaust the combinations of patterns in their arguments, and patterns in

all contexts (functions and case expressions) may be nested arbitrarily deep. This feature greatly simplifies the task of defining more complex behavior. Without it, each matched parameter of a function requires an additional level of nested `case` expressions. The same is true of each level of nesting within a single pattern. Even relatively simple functions can be painful to write, such as `unzip`:

```
unzip = FUN(list ->
  case list of s1 { Nil -> let {res = CON( Pair nil nil)} in res;
                 Cons x xs -> case x of s2 {
                   Pair a b -> case unzip xs of s3 {
                     Pair as bs -> let {aas = CON(Cons a as);
                                        bbs = CON(Cons b bs);
                                        res = CON(Pair aas bbs)}
                                   in res }}})
```

In mini-Haskell this becomes

```
unzip :: List (Pair a b) -> Pair (List a) (List b)
unzip Nil = Pair Nil Nil
unzip (Cons (Pair a b) xs) =
  case unzip xs of
    Pair as bs -> Pair (Cons a as) (Cons b bs)
```

Of course, this complex pattern matching must eventually be transformed into the primitive, nested case expressions. Fortunately there are well-documented algorithms for achieving this, and Philip Wadler's chapter on the compilation of this type of pattern matching proved invaluable [39]. Wadler describes the *match* function which, in its simplest form, produces a correct, but potentially large translation with some redundancy. This function was implemented with only minor modifications to fit the shape of mini-Haskell's abstract form. The implementation is augmented by some light flow-analysis to eliminate some of the redundancy and make the resulting code easier to read. There are further optimizations suggested by Wadler to address the size of the generated code, for example, when multiple branches lead to the same large expression, but the additional complexity outweighs the benefit for a testing language.

Less significant features. Mini-Haskell provides some smaller conveniences that don't warrant individual sections but are worth mentioning.

- *Constructors as functions.* As in Haskell, constructors are first class, acting as true functions. This is implemented by simply generating a function that wraps the appropriate `let`-bound `CON` object in STG.
- *Primitive operations as functions.* Similar to constructors, primitive operations are given first-class status as functions, with roughly the same rules.
- *Lambda expressions.* mini-Haskell supports anonymous lambda expressions for the simple, one-off functions that often find a place in Haskell code.
- *Type annotations.* The ability to assert the type of some variable is often valuable as a sanity-check and provides documentation that the compiler can verify. Mini-Haskell's type annotations (or signatures) behave like those in Haskell, being passed to the type checker as initial assumptions.

– *Haskell Block.* Being a subset of Haskell, it may be desirable to test a mini-Haskell program compiled through our toolchain against the same program (and same source file) compiled with another Haskell compiler. Because mini-Haskell exposes a different set of primitives the Haskell Block was introduced to allow the inclusion of the `MagicHash` extension and the aliasing Haskell's primitives (e.g. `(+#)` and `(-#)`) to the names given in mini-Haskell. This block, delimited by `{-#-}` strings, is ignored by the mini-Haskell compiler, and arbitrary "real" Haskell code can be placed therein to create the same mini-Haskell environment for full Haskell compilers.

C Test suite and testing

We have developed an extensive test suite which uses the CMake CTest tool [33]. The full test suite is automatically run, and any failures reported, after every commit to the Git revision control system. We use a Jenkins continuous integration server to automate this build and testing process [45]. The Git repository is hosted externally so our student collaborators can contribute at their leisure.

There are several hundred small STG and mini-Haskell programs in the test suite, which is partitioned into a number of directories which control what testing is done. Given our goal of exploring various levels of strictness, most tests are run with all evaluation strategies (three degrees of function strictness by two degrees of constructor strictness). Interesting classes of test programs are those that require less-strict or sharing semantics. For these programs we test not only that the test leads to the expected result with less strict semantics, but also that the program returns a *BLACKHOLE* in the case of a stricter semantics. Students who were Haskell/STG beginners have contributed substantially to the test suite. We have also found that the test suite has been of use as a learning tool for these students to discover what levels of non-strictness are required for their programs, a benefit that we did not initially envision. Over time we have developed extensive STG and mini-Haskell preludes of common programming patterns to aid in the writing of test programs.

Another important class of test programs are those that should fail with a known error, e.g., a parsing or typing error. For these programs the test suite checks the output against an expected error regular expression. This class of test programs has proved valuable in development and debugging of both the STG and mini-Haskell front-ends.

We also run each test with various levels of garbage collection enabled. This has exposed bugs in both the code generator and runtime system. Interestingly, in practice garbage collection can both mask and expose bugs.

The various combinations of strictness and garbage collection levels mean that we run a quite large number of tests (currently order thousands). However, this is fully automated via the testing and continuous integration infrastructure.

We have also implemented a simple logging facility which can be controlled at both compile and run time which helps in determining why a test is failing.

References

1. European Exascale Software Initiative, Toward Exascale Computing, http://www.eesi-project.eu/
2. GHC/As a library, http://wiki.haskell.org/GHC/As_a_library
3. Haskell distribued parallel Haskell, Control.Parallel.HdpH, http://hackage.haskell.org/package/hdph-0.0.1/docs/Control-Parallel-HdpH.html
4. International Exascale Project, http://www.exascale.org/iesp/Main_Page
5. Scientific Grand Challenges: Architectures and Technology for Extreme Scale Computing, http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Arch_tech_grand_challenges_report.pdf
6. Strict-ghc-plugin: Compiler plugin for making Haskell strict, http://hackage.haskell.org/package/strict-ghc-plugin
7. The Glasgow Haskell Compiler: Strict & StrictData, http://ghc.haskell.org/trac/ghc/wiki/StrictPragma/
8. Top 500, The List, http://www.top500.org/lists/1993/6/
9. Tying the Knot, http://wiki.haskell.org/Tying_the_Knot
10. Aiken, A., Williams, J.H., Wimmers, E.L.: The FL Project: The Design of a Functional Language (September 1993), http://theory.stanford.edu/~aiken/publications/trs/FLProject.pdf
11. Backus, J.: Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs. Commun. ACM 21(8), 613–641 (Aug 1978)
12. Backus, J., Williams, J.H., Wimmers, E.L., Lucas, P., Aiken, A.: FL Language Manual, Parts 1 and 2. Research Report RJ7100, IBM Almaden Research Center Dept. K53/803 (October 1989)
13. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: Expressing Locality and Independence with Logical Regions. In: Proceedings of the Conference on Supercomputing (SC'12). pp. 1–11 (2012)
14. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Structure Slicing: Extending Logical Regions with Fields. In: Proceedings of the Conference on Supercomputing (SC'14). pp. 845–856 (November 2014)
15. Bernie Pope: Ministg, http://wiki.haskell.org/Ministg
16. Bird, R.: Using Circular Programs to Eliminate Multiple Traversals of Data. Acta Informatica 21(3), 293–250 (1984)
17. Bolingbroke, M., Cheplyaka, R., Mitchell, N.: The Monad.Reader Issue 12: Summer of Code Special (Nov 2008), http://wiki.haskell.org/wikiupload/f/f0/TMR-Issue12.pdf, ch. 3, Compiler Development Made Easy
18. Cesarini, F., Thompson, S.: ERLANG Programming. O'Reilly Media, Inc., 1st edn. (2009)
19. Collins, A., Grewe, D., Grover, V., Lee, S., Susnea, A.: NOVA: A Functional Language for Data Parallelism. Technical Report NVR-2013-002, NVIDIA Corporation (2013)
20. Daly, J.: A higher order estimate of the optimum checkpoint interval for restart dumps. Future Generation Computer Systems 23(3), 303–312 (feb 2006)
21. Davis, K.: MPP Parallel Haskell. In: Draft Proceedings Implementation of Functional Languages (IFL '96). pp. 49–54. No. 1268 in LNCS, Springer-Verlag, Bonn/Bad-Godesberg, Germany (1996)
22. Dongarra, J., et al.: The International Exascale Software Project Roadmap, http://www.nersc.gov/assets/pubs_presos/IESP-Roadmap.pdf

23. Harris, T., Marlow, S., Jones, S.P.: Haskell on a Shared-memory Multiprocessor. In: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell. pp. 49–61. Haskell '05, ACM, New York, NY, USA (2005)
24. Heren, B., Hage, J., Swierstra, D.: Generalizing Hindley-Milner Type Inference Algorithmsa. Tech. Rep. Technical Report UU-CS-2002-031, Institute of Information and Computing Sciences, Utrecht University (2002)
25. Heroux, M.A.: Exascale Programming: Adapting What We Have Can (and Must) Work (January 2016), http://www.hpcwire.com/2016/01/14/24151/
26. Hughes, J.: Why Functional Programming Matters. Computer Journal 32(2), 98–107 (1989)
27. Johnsson, T.: Attribute Grammars as a Functional Programming Paradigm. In: Kahn, G. (ed.) Functional Programming Languages and Computer Architecture. LNCS, vol. 274, pp. 154–173. Springer Verlag (1987), iSBN:0-387-18317-5
28. Lippmeier, B.: The Disciplined Disciple Compiler (DDC), http://trac.ouroborus.net/ddc/
29. Liu, H., Glew, N., Petersen, L., Anderson, T.A.: The Intel Labs Haskell Research Compiler. In: Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell. pp. 105–116. Haskell '13, ACM, New York, NY, USA (2013)
30. Marlow, S., Peyton Jones, S.: Multicore Garbage Collection with Local Heaps. In: Proceedings of the International Symposium on Memory Management. pp. 21–32. ISMM '11, ACM, New York, NY, USA (2011)
31. Marlow, S., Peyton Jones, S., Singh, S.: Runtime Support for Multicore Haskell. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming. pp. 65–78. ICFP '09, ACM, New York, NY, USA (2009)
32. Marlow, S., Peyton Jones, S.L.: Making a Fast Curry: Push/Enter vs. Eval/Apply for Higher-order Languages. In: Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming. pp. 4–15. ICFP '04, ACM, New York, NY, USA (2004)
33. Martin, K., Hoffman, B.: Mastering CMake. Kitware Inc. cop. 2005-2007, Clifton Park, New York (2007)
34. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 3.1 (Jun 2015)
35. Michael, M.M., Scott, M.L.: Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. In: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing. pp. 267–275. PODC '96, ACM, New York, NY, USA (1996)
36. Petersen, L., Orchard, D., Glew, N.: Automatic SIMD Vectorization for Haskell. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. pp. 25–36. ICFP '13, ACM, New York, NY, USA (2013)
37. Peyton Jones, S.L.: Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine. Journal of Functional Programming 2(2), 127202 (April 1992)
38. Peyton Jones, S.L., Launchbury, J.: Functional Programming Languages and Computer Architecture: 5th ACM Conference Cambridge, MA, USA, August 26–30, 1991 Proceedings, chap. Unboxed Values as First Class Citizens in a Non-strict Functional Language, pp. 636–666. Springer Berlin Heidelberg, Berlin, Heidelberg (1991)
39. Peyton Jones, S.L., Wadler, P.: Implementation of Functional Programming Languages, chap. Efficient Compilation of Pattern-Matching, pp. 78–103. Prentice Hall, New York (1987)

40. Pointon, R., Trinder, P., Loidl, H.W.: Implementation of Functional Languages: 12th International Workshop, IFL 2000 Aachen, Germany, September 4–7, 2000 Selected Papers, chap. The Design and Implementation of Glasgow Distributed Haskell, pp. 53–70. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
41. Reinders, J.: Intel Threading Building Blocks. O'Reilly & Associates, Inc., first edn. (2007)
42. Seo, S., et al.: Argobots, https://collab.cels.anl.gov/display/ARGOBOTS/
43. Sheard, T.: A Pure Language with Default Strict Evaluation Order and Explicit Laziness, presented at the 2003 Haskell Workshop, Uppsala, Sweden, Aug. 28, 2003.
44. Simon Marlow (ed.): Haskell 2010 Language Report, http://www.haskell.org/onlinereport/haskell2010/
45. Smart, J.F.: Jenkins: The Definitive Guide. O'Reilly Media, Inc. (2011)
46. Stewart, D.: Multi-paradigm Just-In-Time Compilation. BSc (Computer Science) Honours Thesis, The University of New South Wales (Nov 2002)
47. Stewart, R.: Reliable Massively Parallel Symbolic Computing: Fault Tolerance for a Distributed Haskell. Ph.D. thesis, Heriot Watt University (2013)
48. Sulzmann, M., Chakravarty, M.M.T., Jones, S.P., Donnelly, K.: System F with Type Equality Coercions. In: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation. pp. 53–66. TLDI '07, ACM, New York, NY, USA (2007)
49. The Stellar Group: HPX, http://stellar.cct.lsu.edu/projects/hpx/
50. Tolmach, A., Chevalier, T., Team, T.G.: An External Representation for the GHC Core Language (For GHC 6.10), http://downloads.haskell.org/~ghc/7.8.1/docs/html/users_guide/an-external-representation-for-the-ghc-core-language-for-ghc-6.10.html
51. Trinder, P.W., Pointon, R.F., Loidl, H.W.: Runtime System Level Fault Tolerance for a Distributed Functional Language. In: Selected Papers from the 2nd Scottish Functional Programming Workshop (SFP00). pp. 103–114. Intellect Books, Exeter, UK (2000)