Pure Logic Program Compilation and Improvement (Optimization) Via Programmer-Controlled Transformations

Fourth Generation Languages Done Right

Malcolm Mumme

Abstract

I describe a novel hybrid approach to efficient parallel implementation of declaratively (and formally) specified applications. My approach proposes novel languages that support formal refinement and parallel distributions. I defer the complexity of designing user-level programming language features in favor of designing intermediate languages with specific properties and sufficient generality.

My primary high-level intermediate language (IL), RILL (Refinement-based Intermediate Logic Language), provides the generality of predicate logic for specification and supports arbitrarily powerful programmer-controlled optimizations, through formal refinement. My secondary IL, Ephemeral, supports low-level parallel processing across a wide variety of platform types. A programmer writes a formal specification in a syntactically pleasing 'surface' language, and invokes a surface language compiler, which translates the specification into RILL, and invokes my proposed system. My proposed system starts with the formal specification translated to a RILL program, optimizes this program via correctness preserving RILL-to-RILL code refinements, then translates the optimized code to Ephemeral, invoking the Ephemeral compiler for translation to specific platforms.

My approach to compiling predicate logic programs in RILL is novel in the way it avoids the generally intractable problem of compiling declarative specifications. I take advantage of the fact that existing solutions to many computing problems can be construed as solutions of specific declarative specifications. I additionally exploit the wealth of existing decision procedures for many classes of problems, such as FOL, SAT, ILP, linear systems, etc., using them as bases for refinement methods. The most universally necessary refinement methods should occupy a 'standard library' of optimization procedures, leaving less universal and more advanced optimizations to extended libraries. RILL provides the novel *Refine* construct, which application programmers use to control the application of optimization refinements that transform their declarative specifications to a more executable and efficient form. When compositions of available refinements do not suffice to optimize a particular specification, the Refine construct also provides the means of applying custom optimizations, which may be constructed by the application programmer. Thus, the implementation of RILL is a step toward obtaining high application performance with a general purpose logic programming language.

The proposed low-level parallel programming language, Ephemeral, mostly avoids architectural dependence by making only minimal assumptions about supported platforms. Ephemeral uniquely supports MIMD arrays with tiny individual processors, not having sufficient memory for the entire program, such as I expect will eventually dominate the high end of the MIMD performance spectrum. The possibility of high application performance is retained by allowing distribution, of both data and computation, to be specified in a somewhat topology dependent manner. The geometry of Ephemeral's computational model is physics-based, ensuring performance metrics based on the model accurately bound communication performance of compiled code executing on parallel platforms.

The resulting hybrid RILL/Ephemeral system will support the declarative specification and implementation of efficient parallel applications. Thus, this research moves toward positively answering the question:

"Can we escape the Sequential Prison by programming in a general purpose declarative language?".

1 Introduction

I briefly summarize the state of the practice with respect to declarative specification, and with respect to parallel programming, in both cases describing potential advances with my approach.

1.1 Execution of declarative specifications

From a software engineering point of view, it would be highly desirable to have a compiler that receives declarative specifications as input, and automatically produces efficient code, guaranteed to conform to those specifications. Given that the compiler is correct, many concerns are eliminated:

1 Programming bugs.

2 Performance bugs, security bugs, etc., given that non-functional aspects may be specified.

3 Other aspects of coding, such as reusability, design elegance, coding style, unit testing etc.

Only the specification itself would be the product of human labor, so that only specification-related concerns remain, potentially reducing the cost of software system development by orders of magnitude, assuming that running such a compiler was feasible and efficient.

However, the problem of fully automatic conversion of formal specifications into efficient code will forever remain in the domain of (real or) artificial intelligence. Solutions to specific cases of this problem are approximated through the employment of large crowds of analysts and programmers.

Nevertheless, computer scientists continue to identify and publish solutions for tractable special cases and sub-problems. The proposed methodology potentially supports crowd-sourcing of computer scientists to solve more advanced cases that individual solutions may not cover.

Existing approaches for execution of declarative specifications, of which I am aware, fall into one of these three categories:

1 The specification language is restricted to computable or reasonable problems (prologs, datalog, SMT, constraint solvers, etc.).

2 A heuristic search strategy is used, that is effective in some sub-domain but is not reliable in more general cases. Some allow human assistance (AI research systems).

3 General systems that support manual effort, allowing machine assistance and tactics (this proposal, HOL, Z, Coq, etc.).

These approaches are briefly described below.

1.1.1 Language restriction

Most of these systems do not generate programs from declarative specifications, but actually attempt to (interpretively) execute the specifications directly, and so actually are solvers for mathematical systems.

The *Prolog II* [21] system solves the subset of predicate logic statements that can be expressed as a set of Horn clauses. Technically, this class of statements includes some undecidable statements, however those that are solvable, are solvable via a procedure that resembles the Prolog interpreter. The Prolog interpreter is not identical to that procedure, primarily because the sequential architectures, supporting the original implementations of Prolog, do not conveniently support non-deterministic search. Prolog II implementations typically search one nondeterministic branch at a time, backtracking only if that branch

fails. It is possible for the search for the solution to a solvable problem to fail due to infinite search in a poorly chosen branch. Thus, the Prolog solvability of some statements depends on the choice of which branch to search, which, in Prolog, depends on syntactic matters, such as statement ordering. Parallel implementations[§5.28] of Prolog variants have more complex solvability conditions.

The *Lambda-Prolog*[45] system has similar characteristics, but with respect to *hereditary Harrop formulas*[31] (a generalization of Horn clauses), instead of Horn clauses.

Datalog[24] is a decidable subset of Prolog, enhanced with features for efficient database access.

Prolog IV[22] extends Prolog with arithmetic constraints over rationals, and is capable of solving rational linear programming systems, and also constraint systems involving transcendental functions. The simplex method and interval methods are used automatically, in addition to others, including the search algorithm from Prolog II.

SAT and *SMT Solvers* solve instances of SAT, where atomic propositions may be predicate queries on variables. The set of available predicates is determined by what theories are supported by the system. Most such theories involve "scalar" variables, as opposed to arrays, trees, etc. Although general SAT is NP-complete, many efficient SMT solvers have become available in recent decades. Various subsets of SAT can routinely be solved in much less than exponential time, in spite of the general case. The research community has taken to including SMT solvers as a basic step in some algorithms [§5.1][§5.16][§5.21][§5.24], even in cases where SAT/SMT problem subset is not necessarily solvable in sub-exponential time.

Constraint Programming systems solve similar kinds of problems, but the idea is to propagate information across constraints, by tracking a set of solutions compatible with a subset of the constraints. Constraints are incrementally added to the subset, possibly chosen according to the graph of constraints and variables, modifying the tracked set of solutions according to the added constraints.

At this point, it should be noted that the language of predicate calculus, extended with the theory of reals and integers, effectively forms an extended logic programming language having a superset of the queries of all the above systems. Continuing along this line, notice that the following systems also have queries expressible in this extended scheme.

Some *Temporal logics* such as *LTL*, that express path queries on sequences of state predicates, over a given state transition graph, are in this scheme. CTL^*, which has a finer quantification scheme, also falls into this scheme if we use higher-order predicate logic. Of course, we must consider some encoding of the given graph to be part of the query, in order to consider temporal logics to fall within the general scheme. In both cases, algorithms have been devised to answer such queries for finite graphs.

Linear systems obviously fall in to this scheme, and have such a restricted form that specific algorithms exist to efficiently solve such systems in polynomial time. Many special cases, such as *symmetric linear systems* and *sparse linear systems* have been analyzed, and are found to be solvable by relatively efficient algorithms compared to the general case. Specific techniques[§5.25] also exist for generating efficient solution code for such systems. However, I do not know of any logic programming system being used to efficiently solve linear systems.

Mathematica is a system for performing symbolic mathematics under interactive manual supervision, and includes representations for a wide variety of standard mathematical problems and logic problems. Mathematica (and some other symbolic mathematics systems) is capable of solving SAT problems, linear systems, a variety of subsets of the queries handled by the above mentioned systems, and, of course, many other types of mathematics problems, such as symbolic integration and algebra problems. Note that Mathematica handles queries which can also be considered a subset of the general scheme.

An *Algorithm* is usually a procedure for efficiently solving a certain subset of queries from the general scheme, where the query has a very specific form, often of practical interest. Similarly to temporal logics, we must consider some encoding of the "input data" to also be part of the query, to make these procedures fall within the general scheme.

Here, we see that, from a certain point of view, all algorithms may be considered to provide solutions to subsets of our extended scheme of logic programming problems, mentioned above. Most of these subsets, however, are not specifically recognized by any logic programming system. This is reasonable, since the set of known algorithms is continually growing, and algorithm researchers do not routinely provide automated methods of recognizing when a logic program is among the set of queries handled by their novel algorithm. This is reasonable, for technical reasons, and because there is no single standard language for expressing all queries.

It is important to note here that there is a general scheme, into which most queries fit, and several restricted schemes, some having a query set that is a subset of the query sets of others. Prolog II, for example, only allows queries that are also allowed by Lambda-Prolog. The more restricted schemes have increased decidability or more efficient solution procedures than the less restricted schemes. The general scheme, and larger sub-schemes are generally undecidable, while some sub-schemes, such as SAT, are decidable, but generally inefficient, and some more restricted schemes, such as 2-SAT, have efficient solution procedures. A few implemented systems, such as Prolog IV and Mathematica, operate on multiple schemes, sometimes achieving efficient solutions, and at other times at least achieving solutions, even though their most general schemes are undecidable.

1.1.2 Heuristic search strategies

AI automatically includes the problem of efficient automatic execution of declarative specifications, since it is unsolved. I am only vaguely acquainted with the existence of such systems. Investigating these systems further would be a questionable use of my time, as I do not believe such a study would contribute significantly to the success of this project.

1.1.3 General manual systems

The Mizar[29] system, based on classical logic and Tarski-Grothendieck set theory, although not actually a system for supporting specification, is worthy of mention, due to its age (39 years, presently) and cumulative volume of use. It is capable only of verifying proofs written in the Mizar language. Mizar is written in plain text files and resembles standard mathematical notation. The Mizar Mathematical Library contains over $52,000$ theorems that have been formally verified and then informally reviewed.

The HOL[65] system, based on higher-order classical logic and set theory is also a theorem proving system, but is intended to support formal specification and verification. Several variants and interactive tools exist to support them. Some variants support tactics, as well as the use of "large steps" (each equivalent to the solution of some problem from category 1) in interactive proof, for user convenience (or confusion, depending on the outcome).

The Coq[8] system has a dependently typed functional programming language and interactive theorem prover. In principle, proofs are done manually and checked by Coq, however, the system evolved to include many automated proof procedures. Coq is notable for the capacity to generate programs, from the constructive proof of the theorem associated, through the Curry-Howard isomorphism[33], with the type of the program.

The Ωmega[57] Haskell-like functional language takes the opposite approach, and allows the programmer to write the code. The programmer may also encode, via the Curry-Howard isomorphism, the correctness specification of the program in its type, which must be checked by the compiler.

Agda[§5.7] takes a similar approach to that of Ωmega, but drops some of the Haskell-like type system complexity, and adds some linguistic conveniences[§5.8] of its own, apparently benefitting software engineering.

The current proposal, similarly to the other systems in this subsection, supports the manual writing and automatic checking of both programs and proofs. As described in Section 2, I use a declarative language for both programs and proofs. Instead of using the Curry-Howard isomorphism to relate proofs and programs, I require the (final) program to be generated as an implicant implying the specification, via an *optimization procedure*, which must be a refinement relation (Section 2.1.2), supplied (potentially) by the programmer. Of course this does not rule out the use of tactics in a supporting environment.

The current proposal also provides a framework that takes advantage of my observations in Section 1.1.1, but without requiring restriction of source code to a specific decidable subset of my language. I intend to build a system where a programmer supplies a logical specification, written in RILL, and directs the compiler to use specific optimization procedures, to transform and refine the specification into a specific RILL subset, that can be recognized as directly translatable to an executable target language.

I envision a future library of optimization procedures, organized in a somewhat hierarchical fashion, where many different subsets (some analogous to computational categories in Section 1.1.1) of the RILL language have corresponding optimization procedures. In some moderately complex cases, an optimization procedure can be thought of as a code generator for the query set of the corresponding language subset. In some very specific cases, an optimization procedure merely recognizes a very specific set of queries, and then substitutes an algorithm that provides the required computation. Some optimization procedures (similarly to tactics) hierarchically invoke others in a specific organized manner, to provide choices for the optimization phase ordering problem. Some very complex optimization procedures may be able to recognize large classes of specifications as transformable to executable form, and may themselves employ complex logic and heuristic methods associated with optimizing compilers and symbolic mathematical solvers. Finally, in cases where no combination of existing optimization procedures suffices to produce executable code, but the problem is actually solvable, the programmer manually transforms the specification, resulting in executable code. S/he then documents this process by producing a novel optimization procedure, that recognizes relevant specifications and transforms them correctly to executable code.

The invocation of optimization procedures uses a novel language construct, that also documents the process and allows the results of optimizations to be inspected, and retained as part of a modified source code.

1.2 Low level parallel programming

Any highly-scalable and efficient model of parallel computation must support a notion of computational space-time corresponding to the physical space-time in which computational artifacts reside.

Today, however, the majority of parallel hardware platforms only support cost models that do not conform to the above statement. The cost of parallel computation is dominated by the cost of communication, yet most existing platforms attempt to hide latency (for programming convenience), and do not guarantee that the latency of inter-processor communication has anything to do with the distance between the source and the destination. Sometimes, the price paid is that every communication has the same (or same order of magnitude) latency as the longest distance communication. This is another legacy feature leftover from the days when processor cycle times were comparable to memory access and communication times.

Although parallel algorithms have been a successful research topic since well before the microprocessor revolution, academics and industry leaders alike have intoned that parallel programming is extraordinarily difficult, pointing to many failed attempts at general purpose parallel processing as evidence.

It appears to me that this is a "straw man" argument, and ignores the many successes which can serve as examples for future attempts. It appears that most extant attempts fall into one of these categories:

1. Successes that are resisted:

SIMD and array computing (for example) has the appearance of being difficult and unfamiliar. Just as an APL keyboard contains many frightening characters which fail to catch the interest of most programmers, the SIMD programming model is not familiar to most programmers, and only had platforms that were not commonly available until the advent of GPU computing. The SIMD model of computing is sometimes incorrectly maligned as being intrinsically less powerful than MIMD models. The use of certain entities, such as vector registers, in array computing, is also unfamiliar, and rarely taught in standard CS curricula.

2. Likely successes that were aborted:

   I know a few examples of promising developments that were cut short due to financial issues. The Transputer[5, 41] project was abandoned after INMOS[55] was sold to EMI, which had no interest in parallel processing. This was still surprising, since the Inmos T9000 Transputer appeared to be vastly superior to other processors of the time. GAUSS (my project at Hughes Aircraft) was abandoned when another relatively applied IR&D project desperately needed additional funding.

3. Potential successes, currently under way:

   Many research efforts[§5.17][§5.30][§5.32][§5.34][§5.39] discussed in Section 5 show promise of providing useful scalable speedups for parallel applications.

4. Failures due to incremental thinking:

   Co-processors, dance-hall architectures, hyper-cube equivalent topologies, super-scalar processing, shared-memory SMP, and ethernet clusters are all partial successes that eventually encounter scalability problems. Parallelizing compilers, SIMD computing models, client-server/RPC-based parallelism, time-warp processing, task/fork/cobegin-based parallelism, and multi-threading are partial successes that are limited by their usual attachment to languages that were designed for sequential execution, in the hopes of resuscitating legacy codes. Most partial successes fall into this category, which I discuss below.

The partial successes, mentioned in the last category above, all have an interesting incrementality property. In each case, the attempt can be construed as a least deviation from standard practice, that might be imagined to provide benefits of scalable parallel processing (sometimes by abusing approximate rules of thumb in order to ignore certain realities, that actually will limit scaling).

Co-processors, client/server and RPC-based parallelism all attempted to hide parallelism within application-specific accelerators or separate computations. The associated latency could be hidden if the operating system could suspend the client or caller so the CPU could be used by other processes. In principle these are very scalable, as long as the co-processors, or servers, are implemented with a sufficiently scalable technique. In practice this is a limitation, since it merely transfers the burden of parallelism to the co-processor or server implementer. We also observe here a slight dependency on the use of a 'normal' operating system in a 'normal' multi-tasking environment, for latency hiding.

Similarly, SIMD and array computing can be thought of as special cases of co-processor or RPC-based parallelism. The SIMD programmer is allowed to write sequential code as before, with the only difference being the use of array libraries, instead of hand-crafted looping constructs. This works quite well when the libraries are appropriate to the task, and the programmers, similarly to APL programmers, are able to envision the desired computation in terms of the available parallel array operations.

In principle, a SIMD computer with locally indexable memory is capable of any computation. Implementation of many control streams on such a computer would be prohibitively tedious, however, since the general case involves the use of interpreters. Conveniently for GPU computing, conditional execution mechanisms suffice for small numbers of control threads. Thus, SIMD architectures are not fully suitable

for general purpose parallel computing, although the current expansion of GPU[§5.35] computing hints that they are still worth considering for many purposes.

Parallelizing compilers attempt to discover possibilities for parallelism through analysis of code written in sequential languages. Such heroic compiler efforts are necessary in cases, such as for legacy code, where the lack of understanding limits attempts at manual extraction of parallelism. Such efforts are necessarily heroic, and likely doomed to failure, since they depend on compilers understanding what programmers do not. Of course, this may work for code that is exceptionally well structured and easy to analyze, which is rarely found.

Super-scalar processors work with unmodified sequential machine code and are suitable for limited parallel execution of legacy code. They attempt to execute nearby instructions in parallel, by understanding each specific execution of those instructions as it is about to occur. This form of understanding is actually simpler than that required by parallelizing compilers, and consists primarily of determining when instructions are ready to issue, subject to hazard constraints. The real-time nature of this understanding requires extensive hardware support that is very invasive to the processor design. The special hardware required does not scale well to issue very many instructions. This form of parallelism has enjoyed considerable commercial success in spite of limited scalability.

Task/fork/cobegin-based parallelism, and multi-threading are ways of specifying parallelism that are based on concurrency (as opposed to parallelism) and on languages that are traditionally implemented in sequential processors. These approaches add a few parallel constructs, such as co-begin or fork(), to an existing sequential language. A global shared memory model is usually provided, to retain the semantics of the sequential base language, and retained in parallel implementations, for compatibility with legacy code. In typical implementations, the concurrency constructs have high execution overhead, as they typically involve operating system service calls. The need for efficiency in sequential implementations forces these systems to have many additional compensatory constructs or services, so that the implementation becomes quite complex. This produces additional inefficiency in parallel implementations, and limits software flexibility and parallel scalability. Typically, these systems are highly dependent on the use of a 'normal' operating system in a 'normal' multi-tasking environment, and cannot be conveniently implemented on architectures with very little storage per processor.

Dance-hall architectures, hyper-cube equivalent topologies, and shared-memory SMP are means to implement the illusion of uniform access to global shared memory required by those software models. In the early days of computing, when processing dominated the cycle time of processors, and memory access was relatively rapid, it was reasonable to consider multiplexing a memory unit between multiple processors. Successive memory systems, however, did not speed up as rapidly as processors, consequently parallel processors require parallel memory banks. In dance-hall architectures, a crossbar (or other equivalent network) connects the collection of processors on one side to a collection of memory banks on the other side.

Thus, the same model, from sequential programming, of uniform memory access cost is retained for the programmer who is responsible for parallel performance. This convenience has a high cost for parallel scalability, because of both coherence issues and physical issues. These architectures are subject to hotspots, and cache coherence problems, which are partially alleviated through the use of snooping and fancy caches in SMP. With increasing numbers of memory banks and processors, these coherence issues force the basic mechanisms to endure increasing delays due to increasing circuit complexity. Additionally, all the above considerations ignore the problem of physical wire delay, and physical volume of processor, memory, and wire. Scaling to large numbers of processors requires a large volume of processors, so that some connections between processor and memory will necessarily involve lengthy wires, or lengthy paths through a network. This causes the basic memory access time to unavoidably increase with scale. With increasing scale, the volume of wires eventually dominates the system volume, so that the paths will be

much longer even than one would expect given the above.

With many of these forms of parallelism, features specific to sequential computing are retained, either implicitly, as is the case with the use of sequential address spaces for shared memory, or explicitly, as with approaches that attempt to preserve legacy code, or that provide parallelism as an added feature to an otherwise sequential language. The retention of these features eventually limits the scalability of the approaches, as described above. Thus, we see evidence of the "Sequential Prison" [§5.36], which detains any attempt at general purpose scalable parallel computing, if it is based on retention of sequential programming concepts.

Similarly to the above partial successes, the so-called "multicore era" is a phase on the route to the efficiency of true massively parallel processing. The "multicore era", will quickly morph into an era of much simpler many-core processors with less dependence on the illusion of a traditional memory structure. The current form of multicore processors will likely survive in applications to battery-powered devices or to legacy codes.

Of course, this doesn't mean that multicore processors, as we currently know them, will abruptly decline in use, any more than hand-knotted rugs abruptly disappeared upon the invention of automated Axminster looms.

I believe that avoiding the sequential prison requires consideration of simple yet physics-based notions of the cost of computation, and abstract architectures that support these notions. I have implemented a language, *Ephemeral* (Section 2.3), which supports such an abstract architecture. Unfortunately, the type system in the current version is too limited for practical use.

1.3 RILL and Ephemeral

The proposed RILL language is based on predicate logic, and thus evades the Sequential Prison, as its constructs are mathematical, and entirely outside the realm of traditional sequential computing.

The proposed Ephemeral language (explained in Section 2.3) evades the Sequential Prison, not by providing novel parallel constructs, but by providing neither sequential nor parallel composition. The sequential or parallel relation between Ephemeral events is not determined directly, but implicitly, entirely by the space-time interval between them.

I propose to implement a new version of the Ephemeral language with an improved type system. The Ephemeral language will support low level programming of a variety of parallel processor types, including those having only a very small amount of local storage per processor, such as those I expect will eventually dominate the extremes of high-performance computing.

The RILL language will have a set of predicates for specifying data distribution in a manner compatible with Ephemeral. The RILL compiler will (after optimization) recognize a subset of the language that intersects the semantics of Ephemeral programs. Compilations will be completed by translation of optimized RILL code to Ephemeral and compilation of the Ephemeral code.

1.4 Background

In the appendices (Section A.1 through Section A.2) I describe, in some detail, the background necessary to understand the project in full mathematical detail, and to understand the technical arguments motivating this project. For the purposes of understanding only the proposed contribution, the reader who is familiar with lambda calculus may read only Section A.2, Lambda Predicate Logic, which describes the core mathematical language of the project.

The main contribution (Section 2) may be understood even without reference to Section A.2, as long as one trusts that the author's arguments are mathematically sound, and knows that the author's "Lambda

Predicate Logic" (Section A.2) is close enough to standard predicate logic that they may temporarily be assumed to be semantically identical. Section 2 has no examples using the language of Lambda Predicate Logic, so detailed understanding of its (abstract) syntax is not necessary. It will be useful to note that the (abstract) syntactic category *Sentence* is used in the obvious way, for sentences (propositions), and the category *Predicate* is used in the corresponding equally obvious way. Lambda Predicate Logic is a typeless 'pure' variant of predicate calculus, where all objects are predicates.

2 Proposed contribution

Here I give the details of the novelty of my contribution. I propose to show that use of RILL-based surface languages[1], especially with Ephemeral, allows highly reusable efficient parallel application programming.

2.1 Extended Predicate logic programming: RILL

The RILL intermediate language is the language of Lambda Predicate Logic extended with the Refine construct and a few built-in *Predicate*s. RILL supports arbitrarily advanced programmer-controlled code optimization by means of the Refine construct, while RILL supports parallel, distributed, and real-time programming by the use of space-time location predicates, used to specify both complex interfaces to external components, as well as distribution of data representations, and hence computation.

2.1.1 Ephemeral Intersection Subset

The Ephemeral intersection subset (EIS) of RILL is a language whose semantics is included within the semantics of both RILL and Ephemeral. The EIS, however, does not include the entire subset of RILL whose semantics intersects with the semantics of Ephemeral, but only a syntactically recognizable subset of RILL. This allows the recognition of EIS to be a feasible problem. Any program written strictly within EIS, not using the *Refine* construct (Section 2.1.2), will be translated by the compiler directly to Ephemeral code, so the compilation will be finished by the Ephemeral compiler. Developments that use the Refine construct must ultimately result in EIS code, and will also result in completion by use of the Ephemeral compiler. Such developments not resulting in EIS code are considered incomplete, and will not finish compilation.

2.1.2 Refine Construct

Although there are many ways to organize the code of a program in RILL, I assume that programs usually comprise a conjunction of many clauses (or *Sentence*s or conjuncts). Thus, in the following, the phrase 'code fragment' means a subset of conjuncts within a conjunction, this subset being viewed as a distinct conjunctive *Sentence*.

The *Refine* construct, in completed form, documents the optimization of a (*Sentence*) fragment of RILL code. It has the following three components.

1. The *Specification*, a code fragment that defines the required behavior of the compiled code.
2. The *Implicant*, a code fragment that refines the specification, and defines the "actual" behavior.
3. The *Refinement Relation*, a *Predicate* that relates the encodings of the specification and the implicant, and refines entailment, guaranteeing that the implicant refines the specification.

[1] The term surface language means the language used directly by the programmer.

The Refinement Relation is usually an Optimization Procedure that produces the encoding of the implicant when given the encoding of the specification. Thus the Refine construct documents the procedure by which a code fragment was refined. In practice, most Refine constructs, as they occur in source code, will contain an empty implicant, which will be completed automatically, by applying the refinement relation to the specification.

Thus, the Refine construct implements a form of meta-programming [§5.2] suitable for refinement of specifications. Various other languages, including C++, Agda[§5.3], Pcf_{DP} [§5.4], Archon[§5.5][§5.6], Scala[§5.14], and especially Scala-Virtualized[§5.15], provide meta-programming features, often as a means for the user to generate complex efficient code, and sometimes as a side effect of polymorphism features.

The Refinement Relation will typically be a reference to some member of a "standard library" of optimization procedures. Typical "standard library" optimization procedures might include low level optimizations, such as constant folding, code motion, and CSE[2], as well as much higher level optimizations, such as *listless transformation*[63], and even much more advanced transformations[§5.20].

The implicant and the specification (as seen by the refinement relation) are *Sentence*s relating the inputs and outputs of the code fragment. The body of the implicant must logically imply the body of the specification, for all inputs and outputs. The refinement relation guarantees this property by also supplying a proof. The implicant and the specification *Sentence*s are encoded into (and/or decoded from) *Predicate*s for inspection(and/or after generation) by the refinement relation.

The Refine construct may, of course, be used recursively in a hierarchical manner, so that they may be applied to parts of the bodies of specifications, implicants, and refinement relations. Thus, a refinement relation may also insert Refine constructs into the implicant it generates. To avoid potential confusion, I define the meaning of a completed Refine construct to be the same as the meaning of its implicant. I do not define the meaning of an incomplete Refine construct, since even a refinement relation may, in fact, be nondeterministic. In this case, there is a class of meanings, which is the same as the disjunctions of the classes of meanings of all implicants that may be produced by the given refinement relation.

Ignoring details associated with separate compilation, a compilation proceeds as follows. The entire program is parsed and typechecked, and converted to an encoding (abstract syntax tree). Then the entire program, as a code fragment, is processed to complete all Refine constructs. This "completed" program is also returned to the environment for use by other tools and, after translation back to surface language, may be made available to the user for inspection, similarly to [§5.19]. The completed Refine constructs are then replaced by their implicants. Finally, if the resulting program is in EIS, the program is translated to Ephemeral and the Ephemeral compiler is invoked to complete the compilation, otherwise the compilation fails.

Processing a code fragment to complete all Refine constructs proceeds as follows. For each non-nested (outer) Refine construct, its specification and its refinement relation are processed to complete all Refine constructs, then, if its implicant is not given, the implicant is generated via the refinement relation, otherwise the given implicant is checked to determine that it is generated via the refinement relation, after which, the implicant is processed to complete all Refine constructs.

Note that there may be considerable implicit compile-time parallelism in the processing of Refine constructs, and that this parallelism may be tempered by various dependencies.

Note also that the compilation process may be made arbitrarily complex (or even non-terminating) by the choice of refinement relations. A RILL compiler may contain a RILL interpreter, for use in applying refinement relations to specifications, alternatively, it may, during the compilation process, complete the compilation of a refinement relation, executing the resulting code during the enclosing compilation. In failed compilations where non-deterministic refinement relations were used, it may be reasonable for the

[2] These optimizations are more relevant to imperative code, but are listed to give the reader a general notion of what is meant here.

compiler to backtrack, in an attempt to ultimately produce a completed compilation. I have the opinion that refinement relations should normally be functions. I do not presently believe compilers should be required to backtrack, and I believe any compilation that backtracks should be required to at least produce a warning. One weakness in the current proposal is that there does not appear to be a simple way for the programmer to control the nondeterminism that may occur in library optimization procedures. These choices should be revisited during the course of the proposed research.

I envision a future "standard library" of optimization procedures, having sufficient optimizations available to give the programmer complete control of code optimization in situations requiring careful optimization for performance. I also envision extended libraries of optimization procedures, possibly resembling techniques from [§5.29], [§5.30], [§5.39], or [§5.42], that go far beyond standard optimizations, permitting programming in improved styles that would not be otherwise feasible. Although populating such libraries is beyond the scope of this proposal, I describe part of my vision here to provide clarity and substance.

The standard library should include all the following:

1. Logic programming equivalents of common imperative optimizations.

   This would include constant folding, code motion, inlining, strength reduction, etc. In many cases with logic programming, the effect of some of these optimizations is achieved automatically via the way code is translated to executable language, or the way code is executed, but there are less obvious cases where these optimizations do apply to logic programs. Even optimizations detrimental to parallelism, such as CSE and strength reduction, are sometimes useful.

2. Optimizations whose correctness depends on statements having programmer-supplied proofs.

   An example would be guard removal from an index operation, where no state-of-the-art prover could automatically determine that the index was within bounds, but the programmer is able to supply a proof.

3. Parametrized optimizations.

   Inlining, where the programmer indicates which references should be inlined, is an example. Another example would be optimization for commonly used values, where the programmer indicates the variable, and the value of that variable, for which the code should be optimized. A more interesting example is a procedure that inserts chosen Refine constructs around parametrically indicated code, to avoid cluttered specifications. Here, the programmer avoids cluttering a specific region of a specification with Refine constructs, by using the 'Refine construct inserter' in an outer Refine construct, so that the outer Refine construct will insert the desired Refine construct around the indicated code in the implicant, leaving the specification less cluttered.

4. Parametric scheduler components for other optimization procedures.

   This would include at least these components:

   - Sequentially applying two other parametricly designated optimization procedures (The implicant of the first procedure becomes the specification of the second).
   - Iterative application of a parametricly designated optimization to a fixed point.
   - Concurrent application of multiple designated optimizations, followed by selection of the "best" result.

   Nested use of such scheduler components allows composition of complex optimization strategies resembling those found in some extant compilers.

5. Schedulers with known effective strategies.

   The results of some previously published optimizing compiler research should be provided in the form of optimization procedures that apply a collection of optimization procedures according to a schedule known to be effective for some programs.

6. Optimizations that recognize certain language subsets.

   This probably includes an optimization procedure that recognizes a functional subset of RILL, and subsequently performs functional-programming-specific optimizations.

7. Optimizations that introduce certain algorithms.

   Most algorithms are optimized ways of performing computations that have a simple specification. It is desirable to have automatic recognition of some occurrences of specifications of well-studied problems, such as searching and sorting problems. An optimization procedure could identify some such instances and replace them with implicants involving use of an appropriate algorithm.

An extended library, going far beyond these types of optimization procedures, might also provide optimizations corresponding to all the categories described in Section 1.1.1, as well as many optimizations for currently unknown domains.

A refinement relation, in a Refine construct within a program body, may have a dependence on run-time data, providing a means to specify run-time adaptation schemes, such as those described in [§5.32] and [§5.37].

One weakness of the above definition of the semantics of the Refine construct is that it does not properly account for occurrences within negated clauses. I expect that properly accounting for such occurrences will require a technical adjustment to the definition, although it may be reasonable to merely limit its occurrences to non-negated clauses.

2.1.3 Space-Time Distribution

Translation to Ephemeral requires that the space-time location of all data items must be (symbolically) determined. This is done using the following RILL features. RILL provides a built-in *Predicate*, *Locate*, that allows designation of the space-time point where a data item will be explicitly represented. Thus, one may write $Locate\langle x,\ p\rangle$ to require that a representation of x be located at space-time point p. Using the *Locate Predicate* on all the elements of an array, for example, can specify the space-time distribution of the array. For this to be feasible, operations must also be provided for various geometric and arithmetic manipulations of point coordinates. There is reason[§5.34] to believe that efficient placements may be automatically determined for some classes of code. Extended libraries of optimization procedures should provide such automatic placement techniques.

It is also necessary to introduce concepts for storage/processor management. A type of data structure will be provided to compactly represent a collection of available space-time points, as well as operations for dividing such collections into smaller disjoint collections, and for consuming individual points in such collections.

2.2 RILL compilation system

I envision an interactive compilation system for surface languages that use RILL as their intermediate form. Ideally, such systems allow syntax directed editing of code, and will allow inspection and modification of compiler output (RILL code with completed Refine constructs), using the surface language equivalents. I also envision eventually providing a surface language, specifically oriented to RILL, with an

interactive implementation, having language extension capabilities for the convenience of surface language implementers.

Although surface language implementations are beyond the scope of this work, it is necessary to provide some way to enter and manipulate RILL code. To avoid all complexities associated with surface languages, I propose to merely construct an interactive system for direct manipulation of RILL abstract syntax trees (AST). This system will provide display and editing of RILL ASTs, file storage, compiler invocation, results editing, and finally, execution via the user's choice of either interpreter or Ephemeral compilation.

2.3 Ephemeral

Ephemeral is a physics-based, fine-grained, low level, pure message passing, strongly typed, non-sequential programming language I implemented in 2003. My 2003 implementation is described in Section A.3. Ephemeral is intended to support the propagation of computational activities through regularly connected networks of extremely simple processing nodes. As it is reasonably efficient to simulate such computational networks upon almost any scalable parallel computation platform, it can be expected that Ephemeral programs can be automatically compiled to run efficiently on almost any scalable parallel computation platform. Here, I describe the principles of Ephemeral, and summarize the changes, to the current implementation, necessary to support the proposed project.

2.3.1 Ephemeral imaginary hardware model

The Ephemeral programmer must imagine the physics of computation to be described by the following imaginary hardware model. In this model, some parameters are imagined to be infinite, while others are strictly bounded. The strictly bounded parameters are not specified, and must be assumed to be small but reasonable, as if they are set at compile time to permit the compilation of any specific legal Ephemeral program. Ephemeral divides computational space-time into a (virtually infinite) collection of *place*s. A place is tiny 4-volume of space-time, having very limited (but non-zero) bounded spacial and temporal extents. Each place is capable of hosting a bounded finite amount of computation, as well as a bounded finite amount of data transfer via messages. To provide for uniform reference to places, Ephemeral construes places to occur in an infinite regular 4-dimensional array. As we also assume references to places are conveniently small, this is actually a slight contradiction. The coordinate system used to address places is not specified by the language, but may usefully be imagined to comprise one temporal dimension and three spacial dimensions. Thus, all of computational space-time is filled with a sea of computational capacity. Ephemeral supports the notion that computational places correspond to physical space-time events (4-tuples), so there is a space-time interval relation between all pairs of places, that repeats in a regular manner, corresponding to the space-time interval relation between the analogous physical space-time events. Every pair of places is related by either a space-like or a time-like interval. The time-like interval relation is transitive. As physical communication is bounded to time-like intervals, a message may travel from an earlier place to a later place in any time-like interval. As physical motion between places may require the transit of intermediate places, some intervals may require the use of intermediate places to relay messages. Each message passing through an intermediate place counts against the data transfer bound of that place, as well as against the data transfer bound of the origin and destination. *Connected* places in a time-like interval may transfer a message without the use of intermediate places. The pattern of connected places is also expected to be regular. Messages have bounded finite information capacity.

A computation occurring at a place may depend only on the data available in messages arriving at the place. The only effects produced by a computation are through messages it sends, which may contain

computational results. Messages containing computational results are constrained to strictly-time-like intervals, because their contents might depend upon any contents of any message arriving at the place of computation. In particular, the destination place of such a message must be such that *all* of its extent is strictly in a time-like interval after the *entire* extent of the place hosting the computation that produced the message.

As with data structures in sequential languages, which provide the illusion of creation and destruction of memory, as heap structures, Ephemeral allows 'creation' of places, each of which is typed and supports some specific computational activity at some specific space-time location. Ephemeral places are single-use, so their 'destruction' is unnecessary. The revised version of Ephemeral will also support compact representations of space-time location pools, from which locations may be allocated for 'place' creation.

2.3.2 Ephemeral Types and Data

Ephemeral data types are as follows:

1. Bit: binary or numerical data, parameterized by size in bits.
2. Action: The code for a computational activity, parameterized by the set of types of place where it may be executed, and by a set of obligations it satisfies.
3. Reference: Indicates a port of a place to receive a message.
4. Tuple: An aggregate of data items, each named and typed with any of the above types. Tuples are not nestable, and the only tuples are messages.

There are also Place types, used to designate the messages expected by each type of place. Additionally, there are pool types, used to describe sets of coordinates (addresses) of regions of space-time available for allocation of new places. Pools are not supported in the currently implemented version of Ephemeral. There is naturally a sub-type relation among Action types, as they are parameterized by a *set* of Place types. This induces a subtype relation among Tuple types, and indirectly on Reference types and Place types. These induced subtype relations are not supported in the currently implemented version of Ephemeral.

A Place is typed as a set of named ports, each having some tuple type. When a place is 'created', references to each of its ports are also produced. References are distributed to the senders of messages, and use the references to send messages to the ports of the Place. All messages to ports of a Place arrive at the Place, and its computation occurs. References are single-use, and may be thought of as a simple variant of futures. To enforce this, they cannot be copied or destroyed. After creation, each reference is transferred from one computation to another, until it is used to send a message, after which it no longer exists. Each reference a computation has must either be transferred into a message, or be used to send a message, but not both. A program is considered erroneous if it mistreats a reference by transferring it to a location that does not have a time-like interval 'before' relation to the location of its referent. This would prevent the proper use of the reference, because the reference must be used to transmit a message before the referent receives it.

A port having a Tuple type, having a data item P of an Action type, may be designated as 'active', and may designate P as its activity.

Values of an Action type contain code that executes an action, except in implementations for processors that support large memories and pointers, where an action value may merely point to the code. Ephemeral is intended to support very-small-memory processors where the actual code must be carried in messages.

The computational activity of a Place comprises the following:

1. Message receipt: Each port of the Place receives its message.

2. Execution: Each activity of each active message is executed in parallel, according to its action code.

3. Termination: The Place and its messages cease to exist.

Action code is compiled from the body of an Action declaration, which comprises a set of basic actions.

1. Triggering actions

   Other Action values (from this or other messages) may be triggered, so they also execute in the same Place. This is not supported in the current implementation.

2. Computation of values

   This is normally where the bulk of ordinary computational work is done. Ordinary arithmetic and logic operators are available to specify these computations.

3. New Place creation/allocation

   New Places may be allocated from a pool of locations, producing also a set of new port references. This will often be an entirely symbolic activity, as pools and references will often have a symbolic form.

4. Message transmission

   Some of the above computed values and port references, as well as symbolic entities, are assembled into messages and sent to their targets, using port references.

Experience with the current version of Ephemeral (Section A.3) revealed that the original design of the type system was inadequate to support the intended programming model, and that inclusion of multiple dispatch[§5.9][§5.13] might be desirable. I now expect to remedy this merely through the inclusion of tuples and sub-typing relationships.

2.3.3 Revised$^{(1)}$Ephemeral

Here I briefly describe the proposed revision primarily via comments in the abstract syntax which follows.

Note that I use the term *obligation*, for port references and for pools. Port references and pools may not be duplicated, since they refer to single use entities. Port references additionally must each be used exactly once, while pools may be discarded without being used. Pools may additionally be split into multiple pools. I use the phrase, 'to *dispose*' a reference, to mean either sending the reference in a message, or using the reference to send a message to the referent port. I use the phrase, 'to *dispose*' a pool, to mean either sending the pool away in a message, allocating a place(s) from a singleton pool, or dividing the pool into multiple pools and disposing each resulting pool. In the proposed revision below, subtyping is explicitly designated by inheritance. It is possible that this inheritance annotation requirement imposes too much burden on programmers. If that is the case, it may be worth considering use of type inference to decrease that burden. On the other hand, it must be realized that, within this research, Ephemeral is used primarily as an intermediate step, and is not meant to be hand-coded.

- $program ::= declaration^*$ ← A program is a set of declarations.
- $declaration ::= bit\_decl \mid code\_decl \mid ref\_decl \mid msg\_decl \mid place\_decl \mid action\_decl$
- $bit_decl ::=$bit_decl $name_{type}$ $[name_{parent\_type}]$ $expr_{size}$ ← $name_{type}$ is a type with $expr_{size}$ bits, or with $expr_{size}$ bits more than $name_{parent\_type}$, if $name_{parent\_type}$ is present.

- *code_decl*::=code_decl

  $name_{type}\ name^*_{parent\_type}\ name_{placetype}\ name^*_{field_1}\ name^*_{field_2}\ name^*_{message_1}\ name^*_{message_2}$

  $name_{type}$ names a code type inheriting restrictions from $name^*_{parent\_type}$, and it is restricted to being hosted by a place of type $name_{placetype}$, must dispose all of the obligations $name^*_{field_1}$, and will not dispose any of the obligations $name^*_{field_2}$. A code of this type will also dispose all obligations located within the messages $name^*_{message_1}$ and will not dispose any obligations located within the messages $name^*_{message_2}$.

- *msg_decl*::=msg_decl $name_{newtype}$[abstract][final][obfinal]

  $(name^*_{parent\_type} \mid name_{message}\ name_{placetype})\ \ ([->][\sim]\ name_{type}\ name_{field}$ [symbolic $expr$])*

  $name_{newtype}$ names a tuple type inheriting from each $name_{parent\_type}$ and adding (or overriding) fields with types $name_{type}$ named $name_{field}$. Any $name_{field}$ identical to an inherited field name is an override, and the associated $name_{type}$ must be a subtype of the type of the inherited field. Any field names shared by more than one parent must have a common subtype among them, or be overridden by a ($name_{type}\ name_{field}$) pair. The pair $name_{message}\ name_{placetype}$ indicates the place type, and which message of that place type may be of this new message type $name_{newtype}$. If there are parent types $name^*_{parent\_type}$, this message type inherits the $name_{message}\ name_{placetype}$ pair from the parents, all of which must have the same $name_{message}\ name_{placetype}$ pair. The optional 'final' indicates that the new type $name_{newtype}$ cannot be the parent of any type. The optional 'obfinal' indicates that children of the new type $name_{newtype}$ cannot add or refine any obligations, such as port reference parameters or pools. The optional 'abstract' forbids sending messages of the new type $name_{newtype}$. The optional '$->$' indicates the code field is an activity of an active message. A tuple subtype may not add activities. Thus the set of activity fields is unchanged by inheritance. The optional '$\sim$' indicates the code value in this field disposes of all obligations in the message, and this holds for subtypes of the message as well. The optional 'symbolic' indicates a field that need not have a representation, as the value may be calculated from the values of other fields via the $expr$. The use of a place type name for a $name_{type}$ indicates the (symbolic) value is a pool, of available coordinates, from which places may be allocated.

- *place_decl*::=place_decl $name_{newtype}$[abstract][final] $name^+_{parent\_type}$ ($name_{type}\ name_{message}$)*

  $name_{newtype}$ names a place type inheriting from each $name_{parent\_type}$ and adding (or overriding) messages with types $name_{type}$ named $name_{message}$. Any $name_{message}$ identical to an inherited message name is an override, and the associated $name_{type}$ must be a subtype of the type of the inherited message. Any message names shared by more than one parent must have a common subtype among them, or be overridden by a ($name_{type}\ name_{message}$) pair. The optional 'final' indicates that the new type $name_{newtype}$ cannot be the parent of any type. The optional 'abstract' forbids creating places of the new type $name_{newtype}$. Note that each place type must have at least one parent. Each implementation is expected to provide one or more 'predefined' place types corresponding to each kind of processor for which it compiles. For each architecturally or topologically distinct parallel processor system supported, a distinct predefined place type is associated with a coordinate system, and a set of coordinate variables, appropriate for addressing/indexing the space-time associated with that system. The predefined place type associated with a given place type will be called its *primordial* place type.

- *action_decl*::=action_decl $name_{action}$ $name_{placetype}$ $field^*$ $allocate^*$ $send^*$ ← defines action code.

  Defines an action, $name_{action}$, which may be used as a code value. When the action, $name_{action}$, is executed in a place, either as an action of an active message, or by being triggered by another action, its execution entails each of the following:

  - The action code of each $field$, which must have a code type, will be triggered. This does not mean that a given action may be executed more than once at a particular place. There is no ordering defined among the actions executed at a place. Several actions that mutually trigger each other, for example, will each be executed exactly once in a given place.
  - The values of expressions used in any of the following steps will be calculated.
  - Each *allocate* statement will execute, as described below, allocating a new place from a pool, and creating references to the ports of the 'new' place. Since pools exist only as symbolic values, these executions are symbolic. Any necessary calculation of port references has already occurred as part of the calculation of expression values.
  - Each *send* statement will execute, as described below, sending a tuple of values as a message.

  A special action, named 'main', which does not require a place, is executed automatically to initiate program execution, similarly to some extant languages. The action code of main is permitted to perform some operations not available in other action codes. In particular, main code may directly 'create' pools (but not overlapping pools), and may also refer to action literals (names of action declarations) as constants, and may used them as values of code fields.

- $field$::= $name_{message}.name_{field}$ ← designates $name_{field}$ from message $name_{message}$ in the host place.

- *allocate*::=allocate $name_{placetype}$ $name_{place}$ $expr$ ← allocates a new place, $name_{place}$, of type $name_{placetype}$, from a singleton pool $expr$.

- *send*::=send $name_{message\_type}$ $expr_{reference}$ $(name_{field}\ expr)^*$ ← constructs and sends a message, to the port referenced by $expr_{reference}$, with actual type $name_{message\_type}$. The message is constructed as a tuple of named fields, corresponding to the $(name_{field}\ expr)$ pairs, which must correspond to all the fields in the message type $name_{message\_type}$. The message type $name_{message\_type}$ must be *conformable* to the type of the message type of the reference $expr_{reference}$. The types conformable to a given type T are T itself, and all subtypes of T. Each $expr$ must be of a type *conformable* to the type of the field $name_{field}$ in the message type $name_{message\_type}$. (This requirement is analogous to the matching of value types to the types of formal parameters in a procedure call in procedural languages.) There is an additional requirement for code type fields marked with the optional '∼' in their message declaration. The code placed in such a field must be of a code type that can execute within the place type $name_{placetype}$ of the actual message type $name_{message\_type}$, and is restricted to disposing all of the obligations in the message $name_{message}$ of the actual message type $name_{message\_type}$.

- *let_statement*::=let $[name_{type}]$ $name$ $expr$ ← defines $name$ as $expr$

  As in the previous revision, 'let' statements are also allowed in many places to calculate values, for programmer convenience. Their ordering is irrelevant, and cyclic definitions are generally not allowed. The type of the new value is inferred when the optional $name_{type}$ is not present. The usual collection of arithmetic and logical operators is available for use with numeric values (having 'bit' types), and the trinary conditional operator is available for use with numeric and code values. $name_{place}.name_{message}$ references the $name_{message}$ port of the place $name_{place}$. Constant values are calculated at compile time. To define coordinate pools, the following expression is also allowed:

- *expr*::=pool [$name_{placetype}$] *expr* ← refers to a pool of unused coordinates.

  Each primordial place type defines a set of symbolic coordinate variables which may be symbolically constrained using a pool expression. A coordinate given by a set of values assigned to the coordinate variables is in the referred pool iff the *expr* evaluates to a non-zero value. Thus, the *expr* defines a symbolic constraint on the coordinates contained in the pool. As the compiler must check that pools are not copied, a kind of constraint solver must be employed within the compiler. I anticipate the solver will only solve simple linear equalities and inequalities. Thus, such constraint expressions should only have simple linear constraints, such as are necessary to define polytopes.

2.3.4 Revised(1)Ephemeral program checking

There are several properties of Ephemeral programs that a compiler must check to ensure it is not erroneous. Here I list some of these properties and give a strategy for implementing each of the necessary checks.

1. Types of values must be conformable to types of message fields where they are sent.

   This check uses now-common object-oriented language type checking technology, with very minor wrinkles for Ephemeral. The types of message fields, and the types of values inserted into message fields, are analogous to types of formal parameters, and types of parameter values, respectively. The type of a value may be inferred from the expression producing it, and the types of the data input to that expression. Only typed messages may be produced, so the type of each field of a message is explicitly known from the declaration of the message type. As a value may have any subtype of the declared field type, checking requires that the compiler is aware of the subtyping hierarchy. This is simple, as each type is a subtype only to its declared parents, and to inherited parents. The only complication is with the insertion of action names (defined in an *action_decl*) into code typed fields. The requirements relating code types to obligations require additional checking described below in items 3 and 4. The special requirement, for code type fields marked with the optional '∼' in their message declaration, can be easily checked with information available from the type declaration of the message being constructed, and the type of the code value being inserted into the message.

2. Obligations must be properly disposed exactly once.

   Within the context of any place, each obligation received in a message must have exactly one disposing action, among the activities of the active messages received by the place. This requirement is met when exactly one of those activities (the disposing action) has a code type that includes the obligation among the $name^*_{field_1}$ of its code type declaration or that includes the message containing the obligation among the $name^*_{message_1}$ of its code type declaration. Additionally, to avoid possible conflicts with subtypes, each of those activities other than the disposing action must either include the obligation among the $name^*_{field_2}$ of its code type declaration or include the message containing the obligation among the $name^*_{message_2}$ of its code type declaration. These requirements are easily checked.

   An action A may dispose an obligation either (1) by performing the disposal itself, or (2) by delegating the disposal to another action it triggers. The possibility of delegation produces a risk of allowing cyclic delegation where the obligation is never actually disposed. This risk is removed by additional rules as follows: In case (1), the action code must perform the disposal, in a way which complies with requirements for reference and pool disposal in items 3 and 4, respectively, and must prevent possible delegation, by not triggering any code field that does not list the obligation in the $name^*_{field_2}$ (or implicitly in the $name^*_{message_2}$) of its code type declaration. In case (2), the disposal must be delegated to exactly one triggered action D, which must have a code type indicating that it disposes

the relevant obligation. Additionally, an assignment of ranks, decreasing with each triggering, to code fields of messages (omitted here for brevity), together with a requirement that rank 0 actions may not delegate, ensures that cyclic delegation does not occur.

3. Obligations may not be copied, and some obligations (references) must not be destroyed before use.

   An action which disposes a port reference, without delegation, either moves it to an outgoing message, or uses it to address an outgoing message. In either case, it suffices to check that the $send^*$ component of an action declaration contains exactly one mention of the disposed port reference, which is not a difficult task. An action which disposes a pool may not copy the pool, but checking this can be complex, as is covered below in item 4.

4. Place pools must be partitioned by user code, and checked by the compiler. Pool fields in outgoing messages must be checked to ensure that coordinates are not shared between different pools produced in an action. Additionally, non-'main' actions must not create pools, but may only transfer or partition pools. The set of coordinates of a pool field, in an outgoing message, is defined by the symbolic pool expression in the declaration of that field. The pool expression may refer to arithmetic fields of the message containing the field, as well as coordinate variables associated with the relevant primordial place type. For simplicity, it is assumed that the symbolic pool expression is a logical combination of linear equations and linear inequalities. Thus, the set of pool coordinates is defined by linear equations and inequalities between coordinate variables and computed expressions. The set of coordinates of a pool field, in an incoming message, is defined by linear equations and inequalities between coordinate variables and arithmetic field values. When the computed expressions are sufficiently simple (linear functions of inputs), it is often possible to determine sharing relations between pools. For any pool A, I write $\underline{A}$ to denote its set of coordinates. The requirements on pool fields in outgoing messages must satisfy two checks as follows: (1) For each outgoing pool field Y, there must be an incoming pool field X, such that $\underline{Y} \subseteq \underline{X}$, and (2) For each pair of outgoing message fields, Y and Z, $\underline{Y} \cap \underline{Z} = \emptyset$. Note that two coordinates are not considered the same if they are associated with different primordial place types.

5. 'main' checks. Pool expressions may occur in outgoing message fields, 'creating' pools. In the 'main' action, check (1), in item 4 above, may be omitted, while check (2) must still be satisfied. As 'main' is the only action where action literals may be used as values for code expressions, the 'main' action code must be checked to ensure action literals are used only in code fields of appropriate type.

6. Port references must be used before they become obsolete. This requirement is easily checked at run-time, with an undesirable loss of efficiency. As a run-time check may fail, the use of run-time checks requires the addition of an failure or exception handling mechanism to the language. Such an addition is undesirable, as it would complicate the language considerably. Checking this requirement at compile time is not feasible in the general case, although there may be an efficient way to check in various special cases. Thus this requirement presents a potential research problem to be solved. It should be noted here, however, that Ephemeral code generated by the RILL compiler might safely compile without this check, if the definition of EIS is appropriately adjusted.

It can be seen that the type system has become moderately complex and inelegant. If a better solution becomes apparent, I will prefer to adopt the more elegant solution. This is likely, due to the way the Ephemeral implementation is scheduled (see Section 4).

3 Motivation and technical considerations

This section gives relevant arguments and technical details related to my language design choices. I briefly summarize the section, and then provide detailed reasons for my preference for logic programing, and also discuss advantages and disadvantages of alternatives. I also additionally motivate this research by listing possible beneficial outcomes. Finally, I give more detail about the motivation for certain choices.

Some time before this project began, I was aware that extant programming languages could be improved considerably. Had C++, Java, or even Algol been available (to me) in 1976, I might have been content to leave programming languages as they were, and just use them to write innovative applications, as I had always intended. At the time, I was interested in simulating physical phenomena, computer graphics, and symbolic mathematics. Around 1976, I used the available programming environment (PDP11 16K with BASIC) to implement simple simulation and graphics programs. An early attempt at a generic parser (for Greibach normal form languages) remained on paper. Apparently, one hour per day at a teletype terminal shared with the other students was not sufficient for all my intended projects. I proposed a multi-threaded processor design around 1977. No one else was interested. Around 1979, I attempted to implement a Lisp interpreter in BASIC, but I was thwarted by limited memory size. I was however, successful in writing a usable symbolic differentiation program and simplifier entirely in BASIC. Also in 1979, I chose to heed the warnings[43, 42], proffered by Mead, Conway, and Sutherland, about designing parallel systems (especially SOC) without proper consideration of the physics of computation, at all levels and scales of system design. I immediately applied the principles I learned to the design of parallel algorithms, and obtained a fast parallel sorting algorithm which had the same complexity, though larger by a constant factor, on physically scalable parallel processor systems, as Batcher's sorting networks, which were unknown to me at the time. This and other similar successes, helped shape my view of parallel processing, and profoundly influenced the design of Ephemeral. These experiences, along with many discussions about parallel processing, also allowed me to realize that parallel processor programming was not actually significantly more difficult that sequential programming, but that the difficulty comes from the 'Sequential Prison', which holds back all those who consider parallel programming only through the sequential programming models they have already been taught. Contrary to this (amusingly), I remained content to think of parallel processors as a collection of traditional sequential processors, connected through (possibly multiple) shared memories or messaging connections, where each processor ran a sequential program, that communicated explicitly with programs running on other nearby processors. The arrival of the early INMOS Transputers[55], with their on-chip integrated processor, memory, and communication switch, clearly indicated that even MIMD parallel processor elements could be significantly different from processors meant to be programmed sequentially. I also seriously thought that virtual machines would provide an efficient and convenient way to control the sometimes necessary movement of computational processes through the network of processors. My views on parallel processing architectures have changed since that time.

I first chose to do this project in 1983, partly motivated by interest in supporting efficient virtual machines through verifying or controlling certain temporal and safety properties of user code, somewhat similarly to [§5.1]. I was also motivated by the desire to have a reasonable programming language. I had originally intended to concentrate on implementation of an extremely efficient virtual machine monitor (VMM), so that virtual machines, and deeply nested virtual machines, could be used routinely within applications as a handy mechanism for various purposes. By 1983, I realized that the best performance for virtual machines could only be obtained when the user-level code and the monitor code were guaranteed to have certain safety properties. This realization motivated my interest in use of automated code transformation and formal methods to ensure temporal and safety properties. There did not appear[3] to be adequate programming language support, at the time, for formal proof checking of safety properties of

[3] This apparent state of things was partly an illusion due to my relatively poor access to the literature at the time.

programs, so I realized, in 1983, that proper investigation of formal methods could require considerable time, and significantly delay my intended VMM project. I chose to defer the VMM project (and all other research projects) until after my investigation of formal methods, and in view of the likely duration of the investigation, I chose to seek only methods that effectively resist obsolescence, ultimately leading to the considerations listed in Section 3.6. As I had been seriously interested in parallel processing since 1979, and some were already predicting the end of 'More's Law' performance scaling, I also considered any system not providing good support for parallel programming to be doomed to momentary obsolescence, and decided it was best to avoid the 'Sequential Prison' (a term coined later by Ivan Sutherland[§5.36]). After considering the state of development methodologies, programming languages, and compilers at the time, and accounting for the above considerations, I realized that implementation of a reasonable programming language required the existence of a powerful but sound intermediate language such as the language I propose using for this project. Thus, in 1983, this project began in earnest.

Around 1990, after several years of study and contemplation, I chose predicate logic programing as the preferred paradigm for a compiler's intermediate language. This was partly because logic programing easily allows reasonable forms of nondeterminism, which is useful in avoiding overspecification. This choice occurred only after carefully considering a number of alternatives. Already being familiar with lambda calculus (see Section A.1 and [4, 18]), since around 1976, naturally I first considered the possible use of formal programming methods based on lambda calculus.

I believe Lambda Calculus based on the theory ($Th(\mathcal{K}^*)$) (see Section A.1) was the only (purely functional) form of Lambda Calculus worthwhile considering for this project. This is primarily because $Th(\mathcal{K}^*)$ is the unique maximal fully extensional theory of Pure Lambda Calculus.

For reasons detailed below (Section 3.2), having to do with the necessity of also supporting "ordinary" mathematics, and meta-logic, I eventually decided that pure Lambda Calculus was not actually the optimal core language for this work. Even with the expanded notion of infinite trees as λ-terms, there appear to be at most a continuum of λ-terms. This is a problem if one ever needs to consider something as large as all the subsets of the reals. Many real programs require such entities in the proper analysis of their requirements, and it would not do to intentionally construct a language that becomes obsolete at the exact moment that one first tries to apply it to useful programs. For a while, I considered using set theory and then standard classical logic (Russell and Whitehead), however found that this was an unlikely foundation for supporting meta-logic. I briefly considered using class theory, and then non-well founded set theory (unaware that Aczel was already developing this), but it seemed unlikely that I could develop this area into something useful. Aside from its connection with set theory, class theory potentially resembles predicate logic, which I eventually chose for this work.

The difficulties with the use of Lambda Calculus for logic and mathematics are overcome in many systems of combinatory logic through the arbitrary inclusion of additional elements. In some cases, an "equals" function, (magically) returns a Church-encoded boolean which indicates the equivalence of two inputs. In other cases, some functions are considered to be characteristic functions of sets, and set-theory related functions are included to (magically) allow some otherwise inexpressible statements. These solutions all present additional opportunities for abuse, so that the associated type systems are made more complex by the need to avoid paradoxes. In some systems, types, per se, are not used, and the additional complexity instead comes in the form of seemingly arbitrary restrictions on certain proof rules.

The proposed solution (use of Lambda Predicate Logic (see Section A.2)) also has the disadvantage of added constructs, but also removes constructs, while retaining all necessary expressivity. In particular, logical connectives are added, and the facility for multiple parameters is added, at the expense of removing functions that return values and currying.

3.1 Logic programing and nondeterminism

Software engineering informs us that excessively early binding and overspecification are the root of many evils. In formal specification languages based on logic, and in logic programming, computations may be specified nondeterministically, allowing for later choice of details not important enough to be included in the specification. Specifically, disjunction between non-failing options allows nondeterministic choice among the options.

Functional programming languages could allow nondeterminism via a 'choice' construct, that allows a value to be nondeterministically chosen from a collection of values. In this case, however, such choices must either be made at compile time, or, if made at run-time, must be made deterministically, due to the nature of functions.

Such a choice function could be executed at runtime only if the choice was between two different computations of the same value. In this case, it might be reasonable to execute both algorithms concurrently, using the result from the one that finishes first, and then aborting the slower algorithm. If the intermediate language is a functional language, however, it is inconvenient to express this strategy, without resorting to such constructs as *engines*[32].

RILL programs express the desired nondeterminism via disjunction, and the desired strategy is expressed as a Refine transformation to a conjunction, and machinery for sending the result and halting the slower algorithm.

In addition to the above reasons for choosing predicate logic programming, there have long been indications in published research, that compilers should use a declarative form to store and process information about programs. This is also observed in the current literature[§5.16].

3.2 Unsuitability of Pure Lambda Calculus for (Meta) Logic

As Lambda Calculus is the simplest known (to me) system capable of expressing any computable computation and logic, it would certainly appear to be the best foundation for formal development of this project. Its simplicity was very attractive from the point of view of minimizing the 'trusted code base', or actually, the number of things on which the validity of the development depends. As I did not wish to increase the complexity of the core of my system beyond what was necessary, I considered only the following options for founding my system upon Lambda Calculus.

1. Encoding propositions directly as Lambda Calculus expressions.

   In this scheme, propositions are encoded as expressions that evaluate to encoded (Section A.1.6) Booleans. Encoded Boolean operators would be used as connectives, while predicates would be encoded as functions that returned booleans. Most systems of combinatory logic follow variants of this option.

2. Use of equations between expressions to encode logical propositions.

   In this scheme, all equations between Lambda Calculus expressions are atomic propositions. Conjunctions can then be expressed as equations between encoded pairs. This scheme also conveniently encodes universal quantification over all functions.

Scheme 1 is apparently the most desirable, due to its simplicity. Certain facts, however, make its use as a logical system more complex than one would like. It is, in fact, an undecidable problem to determine whether a given Lambda Calculus function returns an encoded Boolean. Additionally, the computational power of Lambda Calculus allows one to easily construct terms that are self-contradictory according to the encodings. These considerations alone are sufficient to force most systems of combinatory logic to adopt a

type system. Such type systems ensure that constructs are used in a limited way, to ensure well-definedness of propositions and to avoid paradox. Other systems of combinatory logic remain typeless, but instead adopt a relatively complex system definition, so that it remains a difficult problem to determine if a given expression actually encodes a proposition or not. In both cases (with and without a type system), the complexity of a logical system using Lambda Calculus is much greater than that of Lambda Calculus itself. This additional complexity alone makes this option less attractive.

Scheme 2 appeared initially to offer significantly improved expressiveness without significant loss of simplicity. Theorem 1 from Section A.1.7 makes it obvious that this scheme is more expressive than scheme 1, when only pure Lambda Calculus is used. Theorem 2 provides for encoding of conjunctions, while theorems 3 and 4 provide ways of asserting that an expression encodes a Boolean, or a Natural, respectively. This scheme also naturally provides for encoding some universally quantified sentences, since any equation (say $a = b$) implicitly implies another $(\lambda x.(a\ x) = \lambda x.(b\ x))$, via extensionality (Theorem 0), which in turn obviously implies a universal quantification (such as $\forall x : (a\ x) = (b\ x)$). It is easy to show that negation of equations cannot be encoded by equations, as indicated in Theorem 5. Although this is inconvenient, it was still conceivable that a useful constructive system of logic could be devised without negation. I was, however, at length (1985) able to prove Theorem 6, which shows that disjunction of equations cannot be encoded as an equation. This fact dooms scheme 2 to very limited usefulness, since conjunction is essentially the only logical connective universally available. There is a partial workaround for encoding disjunction, which involves using disjoint unions as with the Curry-Howard isomorphism. However, the encoding of disjoint unions must explicitly give the (possibly boolean) discriminant, so this form of union is only available when the actual discriminant is explicitly computable.

An additional problem with both schemes 1 and 2 arises from the simplicity of Lambda Calculus itself. Under the most liberal semantics, pure Lambda Calculus has at most a continuum of entities. The proof of many mathematical theorems, and surely proof of correctness of some practical programs, requires the mention of such entities as all subsets of the reals. The mismatch in cardinality means that there will be statements that hold for all entities in pure Lambda Calculus that do not hold for some entities within subsets of reals. This means that theoretically, pure Lambda Calculus is unsuitable for use with such mathematical analysis. There are workarounds for this problem, such as the addition of UR-elements to define reals separately, or the use of constructive reals and constructive subsets of reals. Some systems use additional elements to define reals, with a requisite burden on the type system to ensure their correct use. Some systems already have a complex type system that effectively includes powersets, already allowing definition of reals etc. Alternatively, the use of constructive reals and subsets must be done carefully to ensure that the proved theorems are meaningful, and also represent an inconvenient limitation of such systems. In either case, the complexity of overcoming the limitation makes the use of pure Lambda Calculus less attractive.

Of course there are other schemes, but I know of none that has the required expressiveness without either seeming arbitrary or having greatly increased complexity. One could, for example, use pure Lambda Calculus merely as a meta-language in which the system is implemented. This is, however, an arbitrary utilitarian choice, and does not involve the actual language implemented in the system.

3.3 (Un)suitability of Set theory

Set theory has many varieties, many of which are adequate for axiomatizing mathematics and for use as a foundation for programming language semantics. The set-theoretic notion of ordinals, for example, provides a natural way of grouping functions based on their termination characteristics, and a reasonable hierarchy of typed functions based on their provable termination characteristics. This is both an advantage, and a disadvantage, since powerful languages based on set theory require type systems to avoid paradoxes.

The required type systems are based, in turn, on the hierarchy of ordinals defined within the theory. The hierarchy of ordinals in a theory is determined, in turn, by the axioms of the theory. Thus, the well-formedness of a sentence of a language of set theory may depend on the set of axioms used. If a theory is extended by an axiom that implies the existence of a 'new' ordinal, the type system of the associated language must be modified to admit sentences mentioning sets of 'new' types, associated with the new ordinal. Thus, the ability to chose a specific language based on set theory, requires that a specific set theory must first be chosen.

The main contribution (Section 2) of this work involves meta-programming (and potentially recursive meta-programming), where a program manipulates an encoding of another program (via the Refine construct), based on analysis of the meaning of the manipulated program. An indirect implication of this type of meta-programming is that the manipulating program must have a higher type (in some ordinal-related hierarchy of types) than the encoded program, if the programs are in a language based on set theory. Consequently, one may not write meta-programs that accept encodings of any type of program as input, but only meta-programs which accept programs of a specific type. This situation is prohibitive to the current work, which involves potentially recursive meta-programming, where a meta-program may have to manipulate encodings of programs having the same type as the meta-program (and in fact may be the same meta-program). The situation would not be as prohibitive if manipulations of encoded programs are based only on the encodings themselves (which may have very low order types). For reasons discussed later, I prefer to have meta-programs assert properties about the meanings of the programs whose encodings they manipulate. These meanings are the functions and other entities of the encoded program, and produces the type dependence that prohibits the use of a set theory-based language.

There are additional reasons I chose not to use a set theory-based language, but they are beyond the scope of this proposal.

3.4 Choice of Lambda Predicate Logic

The rejection of set theory-based languages led to a brief consideration of using class theory-based languages. The apparent complexity of most systems of class theory, combined with the noticeable similarity between set comprehension and lambda notation, along with much though about how to avoid causing obsolescence of programs, lead me to construct the simple intermediate language described in Section A.2. Lambda Predicate Logic combines lambda notation with propositional logic notation, forming a typeless language for predicate logic. The language is intrinsically typeless, and allows self-contradictory sentences, as do some systems of combinatory logic. These systems of combinatory logic avoid paradoxes by not allowing derivation of self-contradictory sentences, guarded by careful choice of derivation rules and axioms. Lambda Predicate Logic can avoid paradoxes in the same way, except that it does not specify a particular logical system, deferring that choice to certifiers responsible for the correctness of deployed systems.

Lambda Predicate Logic is only marginally more complex than the language of Lambda Calculus, permits recursive meta-programming through its typelessness, and allows encoding of set theory through its implicitly infinitary nature.

3.5 Potential benefits of this project

This success of this research could serve many purposes, especially the following.

1. Promotion of Reuse.

   I promote reusability of code by encouraging high-level formal specification and refinement based development through the development and use of reuseable optimization procedures. I have more to

say about this in Section 3.6. Note also that obtaining the full benefits of reusability also depends critically on correctness and on lack of sequential bias.

2. Correctness.

   I promote correctness of compiled code by encouraging high-level formal specification and refinement-based development through the development and use of optimization procedures that refine entailment. I have more to say about this in Section 3.7.

3. Parallel processing/Lack of sequential bias.

   RILL implements an idealized form of refinement-based development in a manner completely independent from the history of sequential computing, while Ephemeral provides an executable target language based on my prediction of the characteristics of the highest performance computing platforms in the forsee-able future. I believe RILL-based surface languages, targeted to Ephemeral, provide the best vehicle for escaping the sequential prison, at least for programmers who have not already been trained to permanently think in terms of sequential computation.

4. Solver improvement.

   The rapid advance of SAT and SMT solver technology has allowed solvers to be included within an increasing variety of research systems, as a convenient component for solving problems that fit within some SMT theory, but which are otherwise haphazardly structured. I see the situation as somewhat ambiguous from a performance standpoint, since the problems solved usually fit within a small subset of the SMT theory used, but the SMT solver doesn't necessarily utilize that fact to provide improved performance. The solver is used merely to illustrate solution by reduction of a problem to an SMT theory problem.

   The application of RILL-based techniques to RILL interpreters, rather than to compilers (via deferring the processing of Refine constructs until run-time, as opposed to the compilation scheme described in Section 2.1.2), provides a framework where optimization procedures may dynamically recognize problems, falling within specific problem subsets, that have improved solution procedures. Thus, when many solvers are described using RILL optimization procedures, composite solvers may be easily constructed, having improved performance characteristics for many specialized cases.

5. Research.

   Currently, there is a steep overhead associated with becoming involved in compiler research. Part of this overhead is due to the necessity of learning to use complex intermediate languages (often having considerable sequential bias) associated with extant compilers. This complexity is minimized with the use of a simple declarative intermediate language such as RILL. Additionally, a RILL-based compiler that provides the surface language with the use of the Refine construct may provide even better surface language facilities for declaratively specifying optimizations, and allow experiments with novel optimizations to be performed without direct use of RILL. This may also decrease the amount of old compiler code one must examine before being able to make any useful contribution, reducing additional overhead. These reductions in overhead may allow more graduate students to be involved in compiler optimization research.

   Much research on choice[§5.12] and ordering[§5.18] of optimizations would find a convenient framework in the product of this research.

   As it is, it seems that optimizing compilers for high-performance systems and for embedded systems are likely to go the way of SQL query optimizers. Production quality SQL query optimizers have

become so bloated, with code to handle all the special cases which must be handled efficiently, that they can only be maintained by large organizations who must maintain them for business reasons. Academic data-base research has largely moved away from SQL query optimization for this and a variety of other reasons. Intermediate languages such as RILL may help research compilers to avoid such a situation, by providing a framework for simpler compiler organization.

6. Crowdsourcing of computer scientists.

   Corporations, having a surfeit of computer scientists, may utilize RILL-based technology for crowdsourcing solutions to certain problems as follows. Compilations that fail, due to lack of adequate optimization procedures, may be automatically stored in a corporate repository of unsolved problems. Problems in the repository may be automatically sent to a pool of computer scientists for bidding, in a manner similar to the operation of Amazon's Mechanical Turk. Solutions may then be stored in the corporate library of optimization procedures, along with procedures for recognizing when the novel optimizations are applicable. Thus, the practical capabilities of the corporation's programmers will improve automatically over the course of time.

7. Industrial reliability.

   With extant compilers, having a fixed set of optimizations, the programmer has no way to ensure the desired performance, other than occasionally resorting to assembly coding. Even compilers that are capable, of the optimizations necessary to provide the needed performance levels, may not recognize the opportunity to apply the necessary optimizations to the code as originally written. Thus the programmer must sometimes resort to unnatural styles and contort the code to force the compiler to produce the desired result. This problem has motivated the use of metaprogramming and other code generation techniques which do not guarantee correctness, and will only become more important as system designers seek to apply ever more advanced and specialized forms of optimization to increasingly larger and more parallel codes[§5.19]. Many papers, such as [§5.25][§5.28][§5.30][§5.35][§5.39][§5.42], illustrate techniques that should be thought of as optimizations or code transformations, yet are sufficiently specialized as to avoid being included in general purpose compilers. In RILL-based languages, however, a programmer may define custom optimization procedures which recognize appropriate opportunities and apply the necessary optimizations as correct program refinements. Thus, the programmer is no longer completely at the mercy of the compiler writer for the generation of efficient correct code.

3.6 Reuse of effort and evasion of obsolescence

Having been aware that it may be some time before a useful system emerges from this research, I chose some characteristics of RILL specifically to avoid various kinds of obsolescence. Here I discuss my proposed solutions to problems relating to various kinds of obsolescence and how using RILL-based languages can promote reuse.

1. Arrogance

   It is naïve to expect any humanly devised system to be immune from obsolescence. Programming languages must inevitably change. A famous computer scientist[4] once said: "Anyone who changes a programming language [compiler], without providing an automatic translator, deserves to be shot!" Therefore, from the point of view of obsolescence evasion, the most important characteristic of a

[4] This was in my memory, but I was unable to track down who said this.

programming language is the ability to translate programs from the language to other languages. Conveniently, intermediate languages are designed to be translated both to and from.

I believe that simple declarative languages that allow direct expression of the intentions of the coder are the most fit by this measure. Programs written in such languages are more likely to usefully survive a change of language than programs written in typical imperative programming languages. Consider, for example, Mizar, a language for expressing mathematical proofs and theories. Although this system was created around 1973, proofs and theories written in Mizar are also accessible in Coq, through use of its 'Mizar mode'. This is in spite of the fact that Coq is based on an intuitionistic logic, while Mizar is based on a strictly classical logic.

There are also other aspects to this problem. Suppose a proof of some theorem, X, makes use of properties of real numbers. Logical system A may define real numbers in terms of Dedekind cuts of rationals, while another logical system B may define reals as a subset of the extraordinals. In either case, the proofs of properties of the reals may be quite elegant, due to various factors. If the proof of X makes clever use of A system-specific definitions, to decrease proof complexity, it may become impossible to translate the proof of X from A to B. The best choice is to choose some abstraction, such as the reals and certain real properties provable in multiple systems, then write the proof of X in a manner that depends only on the abstraction, independently of the way the abstraction is defined in the particular system. Thus, we see that separation of concerns remains an important technique, and that part of the burden of obsolescence evasion remains with the user of the language.

2. Incorrectness: Reusing libraries of composable bad components leads to endless debugging.

   As the age of composing large systems from vast libraries of reuseable components draws asymptotically closer, the cost reduction benefits of software reuse will be only partially obtained, unless library code is proven correct (or programmers start writing only correct code). The reasons for this are simple and unavoidable. The probability of obtaining a correct system is exponentially small in the number of components. Every use and reuse of an unproven component increases the probability of system failure, and exponentially increases the need for system testing. Thus, gains from the reuse of software components are lost due to the cost of additional testing, and to the costs of debugging and system failure.

   The feasibility of reusable components depends critically on the correctness of those components. To keep reusable components from endlessly contributing to an exponential increase in software costs, they must be proven to comply with their specifications. Additionally, the compositions of components must have formal specifications to which compliance is proven.

   RILL promotes correctness of code in an obvious way, by requiring that code be produced through refinement of specifications, and requiring that refinement relations (Section 2.1.2) be refinements of entailment.

   In some ways, this represents a limitation of my system, as code generation techniques such as [§5.24] cannot be supported.

3. Specification changes can make source code obsolete

   While the main burden of reusability rests with the programmer, RILL-based languages can improve the situation considerably. RILL supports a software development methodology that separates the coding task into two parts: (1) formal specification, and (2) optimization development and application. Part 1 (formal specification) primarily comprises analysis and formal codification of informally given requirements, and its products are naturally somewhat reusable, in the case of gradually changing requirements. The production of source code in previous languages and methodologies cannot

rely on the compiler to provide a level of optimization sufficient to allow it to directly process formal specifications. Instead, the programmer must manually optimize the source code, while using a variety of 'software engineering' techniques, attempting to maximize reusability of source code, along with other important characteristics. Compilers primarily provide useful low-level optimizations to compensate for the coding styles, such as OOP, needed to increase reusability. Careful analysis and automatic inlining of small methods, for example, can sometimes help compensate for the indirection overhead often associated with method calls.

For all that, a small change in specification can still render large amounts of hand-optimized source code irrelevant. RILL code is optimized into executable form via optimization procedures applied in Refine constructs. The related programming effort is in the construction of the optimization procedures. A specification with a small change may still be processed by the same optimization procedures, automatically reusing the effort put into their production. Larger changes may require a modification to the choice of optimization procedures, or the production of additional optimization procedures. In either case, the optimization procedures themselves are likely to be highly reusable. Thus, RILL supports a development methodology that permits a very high degree of code reuse.

4. Language dependence.

   It is somewhat less obvious that source code should be written in a way that is as independent of the programming language as possible. Although this is hinted by item 1, my primary motivation for this has to do with meta-programming [§5.2]. [§5.3] addresses a particular aspect of language/encoding independence for the Agda language. [§5.11] addresses making user-customized automated proof procedures less dependent on the specific forms of expressions, and thus is parallel to my work, in the area of proof automation rather than program optimization. Some systems[§5.5][§5.6] appear to avoid this issue in the interest of making the language itself as simple as possible. Other systems[§5.4][§5.14][§5.15] supporting meta-programming do not appear to address such issues. Systems supporting model transformation[§5.23] recognize a related issue, but favor heterogeneity and use of additional languages as adapters.

   RILL depends critically on the use of refinement relations in Refine constructs, while it is intended to be used as an intermediate language for optimizing code written in relatively pleasant surface languages. Since it is desirable for such surface languages to also support a form of Refine construct or equivalent, I must consider carefully how optimization procedures (to be used as refinement relations) are to be written in such surface languages. Refinement relations written directly in RILL will directly access and manipulate encodings of RILL predicates. As the programmers of such languages should not be required to learn RILL, refinement relations in surface languages will not directly access RILL code. Instead, refinement relations in surface languages, and also most refinement relations written in RILL, should avoid reference to the actual encoding of the RILL predicates, and instead should reference the predicates themselves. Reference to the actual predicates more closely matches the intent of most refinement relations, and use of the encodings is simply an implementation necessity. For example, a refinement relation associated with streaming might ask if the inputs and outputs of a predicate are lists, without specifying how to determine this from the encoding of the predicate. In this example, another optimization procedure will be applied to this refinement relation, supplying an analysis algorithm that accesses the actual encoding. Thus, some refinement relations may be specified in a manner that is independent of programming language, by utilizing additional optimization procedures to insert the RILL-encoding-dependent implementation.

   In the absence of meta-programming, relative language-independence is possible with traditional sequential languages by using only features that are common to many programming languages. Such

carefully written code can easily be translated from one programming language to another. Unfortunately, the inability to use special features is likely to produce poor performance in all languages. Use of meta-programming features, such as "EVAL", usually removes the possibility of language independence. A program conveniently written in say, JavaScript, for graphing a user-specified formula, for example, might use EVAL as follows. The formula is input as a string, and a loop starts evaluating the formula for successive graph points. In each iteration of the loop, the 'x' coordinate is calculated, converted to a character string, and inserted wherever 'x' appears in the formula string, which is subsequently passed to EVAL to determine the result to use as the 'y' coordinate. In this process, we see that the user must know JavaScript to correctly write the formula, and that various properties of the JavaScript language, such as its being encoded as character strings, are utilized by the graphing program, making it somewhat language dependent. In this example, the program is not excessively dependent on the language, because it assumes the user knows JavaScript, and thus is able to pass the burden of constructing a correct formula to the user. If it is required that the program not allow the user to 'hack' into it by passing a malformed or malicious formula, the program must then parse and analyze the input formula string, resulting in considerable language dependence, so let's assume this is not a requirement. However, this program is still inefficient, as EVAL must parse and execute the same code many times. The solution to this inefficiency is to use a 'staging' optimization, where the formula string is inserted into a larger body of code that, when evaluated, provides a specialized JavaScript function that produces the entire graph for that function. This more efficient solution is necessarily more language dependent, as it also involves constructing source code for an entire function body from character strings.

Macro-based meta-programming is available in Lisp, and many sequential programming languages, but obviously involves considerable language dependence in most cases. Template-based meta-programming obviously has less language dependence, however common languages (C++) with template meta-programming tend to have complexities that would cause difficulty in translating to other languages.

Thus, RILL may be the first language that supports language-independent meta-programming well.

5. Incompleteness or inconsistency.

   Gödel's incompleteness theorem warns us that any reasonably powerful and consistent system of logic is necessarily incomplete. Thus, in any practical proof assistant based on a specific logical system, some true theorems exist that cannot be proven, hence there are correct programs whose correctness cannot be proven.

   I evade this problem by choosing only a vaguely defined class of logical systems based on equational reasoning, as described in Section A.2.3. The choice of a specific logical system, as described by a set of axioms, is left to the certifier responsible for the safety of the deployed system. As necessary, a carefully chosen logical system may be extended. A succession of incrementally extended incomplete logical systems may asymptotically approach completeness as closely as needed, without being inconsistent.

6. Compiler dependence.

   Due to the semantic distance between pure declarative languages and executable imperative languages, performance characteristics of declarative programs are highly dependent on the quality of the compiler's optimization phase. Many previous language development efforts were able to get by with weaker optimizations, by means of seemingly slight language changes that provided the user with a little more control of the compilation or execution process, to obtain improved performance.

An excellent example of this is the "cut" feature of Prolog, which allows the programmer to cause the interpreter to skip much useless computation. Because this feature is actually, in some sense, an imperative feature, it soils the pure declarative nature of the language. This problem may be 'worked around' by only using the feature when it corresponds to giving the interpreter some true information about the program. The fact that cut may be used elsewhere illustrates a way of lying to the interpreter, which accepts the 'information' without proof. Many other 'pure' functional or logic-based languages have acquired such impure features, which allow the programmer to control performance via some optimization whose correctness is not checked by the compiler or execution system. The inclusion of these features complicates the actual semantics of the programming language. Additionally, the abuse of these features allows the compilers to believe lies about the code, with consequent loss of trust in optimizations based on the compiler's correct understanding of the code.

RILL avoids this problem by requiring optimizations to be introduced only through the Refine construct, and requiring that the actual refinement relations involved are refinements of entailment. Thus, the compiler requires no additional language features for performance improvement, and RILL remains purely logic-based and independent of any specific compiler implementations.

7. Technology dependence of compilers for performance.

   Many extant programming languages are riddled with features intended for efficient implementation on (sequential) processors available at the times those languages were conceived. However, efficient sequential control structures and optimizations often become a hindrance on parallel platform implementations. I seek to avoid these forms of obsolescence by basing RILL on pure predicate logic, and compiling code to Ephemeral, which is designed to be most efficient on a form of processor which exists in the future (see Section 3.8), but is not yet extant.

8. Technology dependence of user programs.

   High-level languages have always held the promise of improving the portability of user programs. To some extent, this goal has been achieved (for simple sequential platforms and GSM SMP parallel systems) by languages such as Ada and Java, which have a well defined semantic(s) that is independent of any particular instruction set architecture. Programs written in these languages can achieve reasonable performance on a wide variety of sequential platforms.

   As the era of massively parallel processing gradually approaches, lack of code portability threatens to again become a very serious problem.

   Code can be expected to become obsolete if it is optimized manually for performance on sequential execution platforms. This is because such code tends to take advantage of sequential processor characteristics that do not permit scalable parallel implementation, especially the presence of a large memory with uniform access time.

   Likewise, parallel code tends to become platform dependent, as it is usually necessary to manually optimize the code for the characteristics of parallel platforms. In particular, processor characteristics such as

   (a) Local memory size relative to problem size,

   (b) Processor interconnect topology,

   (c) Memory access limitations,

   (d) Message latency and minimum message size,

and more, must be accommodated through manual performance optimizations in the source code. Some current research efforts[§5.32][§5.34][§5.37] are attempts to alleviate this burden of manual optimization. This manual optimization makes the resulting codes less portable to other parallel platforms with differing characteristics, resulting in reduction of potential reuse.

RILL-based surface languages provide a framework where optimization procedures may be chosen manually, to maximize performance for any desired platform, while the optimization of the parallel codes is performed by the optimization procedures rather than manually. This greatly reduces the cost of re-optimization for porting code to novel parallel platforms.

Having carefully considered ways to evade all these sources of obsolescence, I consider it possible that RILL might become useful before it becomes obsolete.

3.7 Correctness

There appear to be three viable routes to provably correct programs.

1. Add invariants to an (apparently) working program.
2. Correctly transform a specification into a more executable form.
3. Derive the program automatically from a proof, via Curry-Howard, as with Coq.

Route 1 has attained considerable success lately, due to performance improvements in provers and the computers that support them, and due to the long history of teaching of programming by invariants. However, as we approach the age of massive parallelism, route 1 seems to be of limited applicability, as the vast majority of extant code tends to be written in a manner rather dependent on sequential execution.

Route 3 seems difficult, since Coq produces only functional programs. Although, in principle, any deterministic program can be expressed as a functional program, there are likely to be problems expressing some parallel algorithms as functional programs (especially those involving scatter operations). Another difficulty is that one must first find and prove the theorem associated with the algorithm, to get Coq to produce the algorithm code.

Route 2 has long been advocated by some in the software engineering community, and the theory of deriving sequential programs by specification refinement has been studied for some time[3]. I believe it hasn't received adequate attention from (parallel processing) tool developers, as extant formal tools appear to produce only sequential programs. Most usually require distinct languages for specification, refinement, or executable code, although there is at least one exception[23] (supporting only sequential code generation). Specification refinement has also been employed in circuit design[§5.40]. I have chosen this route, as it appears to have the greatest potential.

3.8 Ephemeral design considerations

The design of Ephemeral is based on my own ideas of how physical parallel processors should ideally behave. As notions of parallel processing are highly varied, it is expected that many might disagree with the ideas upon which Ephemeral is based. Before about 2000, I previously considered compilation of RILL directly to machine code, and had some idea of what the ideal target machine should be like. Considerations related to physics, previously mentioned at the beginning of Section 3, convinced me that the ideal parallel architecture involved a 3-D mesh of some kind of processing element, with approximately nearest-neighbor communications. Many have promoted the idea that mesh designs could only be made 2-dimensional, due to the need to use the third dimension for heat exchange and wiring. I believe such considerations

are ultimately irrelevant, because the 4th-power law of fluid flow clearly provides a way of cooling dense 3-dimensional structures. Additionally, my analysis of parallel algorithms and communication patterns indicated that a 3-D mesh without extra wiring would always perform asymptotically as well as any 2-D mesh with any amount of extra 3-D wiring, and usually would perform asymptotically much better. Physics considerations also lead me to abandon early any thought of using uniform global shared memory addressing, and to doubt the need for hardware support of global shared NUMA as well. Familiarity with various programming paradigms and compiler optimization techniques extant in the late 1980's convinced me that fine-grained parallelism, for both message size (and object size) and task duration, was desirable for improved scheduling of parallel tasks. I believed this was achievable with improved processor designs as long as task switching time could be minimized. The appearance of the INMOS Transputer[5, 41] proved that task switching time could indeed be minimized with a simple processor design. Contrarily to the prevailing preference for large-grained parallelism, recent papers[§5.38][§5.31] suggest smaller task sizes improve overall performance, even for fairly traditional processors. Recent research also provides mechanisms [§5.26] that provide efficient transfer of small messages, even on platforms that only support large messages, and provides simple hardware extensions[§5.33] that support efficient fine-grained synchronization on complex processors. Interest in 'wafer-scale integration' lead to consideration of various fault-tolerance and yield enhancement schemes for fabrics of processors with very many elements. From these considerations, came the realization that the fastest processors would eventually be those whose processor elements contained minimal state. Packaging considerations made the production of such processors appear unlikely in the near future (of 1989), especially by persons such as my self. Aware that hardware-supported multithreading could efficiently simulate such architectures on processor elements that are not as minimal, with the only cost being the loss of some actual parallelism, I chose to continue working with the same general notion of how an ideal computer should be structured. This notion was further refined by realizing that employing processor elements (or simulated processor elements) with minimal state effectively eliminates almost all need for traditional tasking and task scheduling, since the (virtual) processor density could easily be higher than the traditional task density. The idea of doing without an operating system or large memories on a very small processing element is generally dismissed by most programmers as impractical. However, embedded systems are often built with such minimal devices, and recent research[§5.41] shows that even Java applications can run on very small customized embedded processors (<500 FPGA logic blocks and <1K RAM). I did not, however have a very concrete idea about the internal operation of ideal processor elements, except that they must have any capabilities necessary to host applications, and supposed they might be RISC-like.

After realizing that I was not going to build any custom massively parallel processors within a few years, I considered that it may be desirable to compile RILL to other platforms, and possibly many different kinds of parallel platforms. Ephemeral was conceived after the year 2000, and was intended solely as a platform-independent relatively low-level target language for RILL. By this time, I had long realized that the employment of processors with minimal state (and no shared memory) creates problems not present on traditional platforms. One problem I was concerned with was distribution of object codes to the processor elements where they should execute, and another was the processor allocation scheme. My ideas for Ephemeral remained somewhat vague, until I read a 2002 Sigplan Notices article titled "Communicating Reactive Objects: Message-Driven Parallelism"[16], as well as a 2003 series of articles about "Totally Message Driven Computation"[17]. Thoughts on these articles inspired a more concrete design, based partly on the notion that the entire program state should be carried in the contents of messages, rather than in an explicit memory. During the Summer of 2003, I began to think that a simple strongly-typed programming language could be designed to address almost all the issues I had considered. After I implemented the initial version of Ephemeral in the Fall of 2003, as a senior design project, the weakness of the simple type system became obvious, suggesting the proposed revision.

4 Schedule

Here I first enumerate the tasks involved in this research, and then propose a specific schedule for their execution.

4.1 Tasks

I propose the following technical tasks be performed prior to dissertation, in addition to solving any further research problems that arise within these tasks. The proposed schedule[5] for completion of these tasks is shown in Figure 1.

1. Finalize design of RILL meta-programming features.

   I will carefully read (and contemplate) certain references, especially [§5.3] and [§5.4]. This should allow the final design decisions to be made quickly. A RILL reference will also be constructed, starting with material from Section 1.4 and Section 2.1, and updated according to the final design. This task may commence without prerequisites.

2. Construction of RILL-specific editor GUI (RILLGUI) with XML AST tree file interface.

   I will construct the RILLGUI which allows manual construction of RILL abstract syntax trees, and their storage and retrieval via XML files. This serves as the foundation for the following tasks, and so should be designed accordingly. For portability, the RILLGUI should be written in Java, or Scala[§5.14], using Java Swing classes, or other graphical toolkit chosen for portability and usefulness in the following tasks. Task 1 is a pre-requisite for a part of this task.

3. Implement front-end features such as manual code manipulations and proof construction/checking.

   The RILLGUI must be enhanced to allow manual invocation of α-conversion, β-conversion, and equational substitution (see Section A.2.3 and Section A.1.1). Entailment relations between constructed entities should be tracked by the RILLGUI, and stored for later uses. The pre-requisite for this task is task 2.

4. Implement incomplete interpreter/solver.

   The RILLGUI must be enhanced to allow manual invocation of a simple solver, which will be capable of solving some small problems that do not produce rapid combinatorial explosion. It is expected to have similar power to a Prolog II interpreter. Prerequisites: Tasks 1 and 2.

5. Implement middle-end functions including optimization by RILL Refine constructs.

   The RILLGUI must be enhanced to allow manual invocation of RILL compilation, which primarily comprises processing of RILL Refine constructs. The compiler will attempt to process RILL Refine constructs using the interpreter developed in the previous task. Compilations will output the modified RILL code back to the RILLGUI for further manipulation and analysis. Prerequisite: Task 4.

6. Definition of common mathematical and programming concepts within RILL.

   For convenience of writing specifications, definitions of common concepts must be useable from within RILL. These definitions must include encodings for finite data structures, numbers, formulae, and sets, as well as related operations and theories. It is not necessary to prove formal properties, such as completeness, of theories at this time. Prerequisites: None.

[5] When composing this schedule, I expected the proposal defense to occur in Nov 2012. There is a 'Plan B' (Section 4.2).

7. Design EIS subset of RILL.

   The EIS must be explicitly specified and analyzed carefully, to ensure it fits the semantics of both RILL and Ephemeral, and is efficiently recognizable. Prerequisites: None.

8. Implement recognition algorithm for EIS and translation to Ephemeral.

   Completion of RILL compilation requires a code that recognizes EIS, and efficiently translates that subset of RILL to Ephemeral. This task may be sufficiently well defined and simple enough to serve as an MS project or even a CS179 project for someone. Corequisite: Task 7. Prerequisite: Task 4.

9. Implement Ephemeral on a high-performance platform.

   I will implement a compiler from Ephemeral to a suitable language and platform, which must be a platform that offers scaling to at least 1K 'large' processors (CISC with >1G RAM each), or 1M 'tiny' processors, in a system available to me. The implementation will be capable of producing code with performance matching the Ephemeral performance model. This may require an initial careful study of the communication infrastructure of the target platform. Prerequisites: None.

10. Integrate Ephemeral implementation with all aspects of existing RILL system.

    The existing compiler should be modified to complete compilation through the new high-performance Ephemeral implementation. The existing interpreter should also recognize EIS, and execute compiled code when appropriate. The existing compiler and interpreter should be implemented on top of RILL and possibly re-optimized to take advantage of the new high-performance Ephemeral implementation. Prerequisites: Tasks 5, 8, and 9.

11. Choose a simple demonstration application.

    I will choose an application for demonstration of the feasibility of refinement based development in RILL. The application must have a reasonably simple formal specification, while practical implementation must require significant optimization for performance. The relevant technology must be known to me, so that progress is not impeded by the need for excess technology transfer. Corequisite: Task 6

12. Implement application in RILL, with translation from requirements to code via domain-specific optimizations, implemented as RILL Refine constructs.

    I will write (in RILL) the formal specification for the chosen application, and attempt to produce an efficient implementation using refinement based development. This will involve development of a library of domain specific optimizations in RILL, usable as refinement relations within RILL Refine constructs. Prerequisites: Tasks 5 and 11. Corequisite: Task 10.

As opportunities arise, it will be advantageous to also perform some of the following publishing-related activities:

PA: Report on the design of RILL meta-programming features, and their impact on implementation of optimization by RILL Refine constructs.

PB: Report on the experience gained from application implementation.

PC: Report on the high-performance Ephemeral implementation.

PD: Papers on any additional research problems solved.

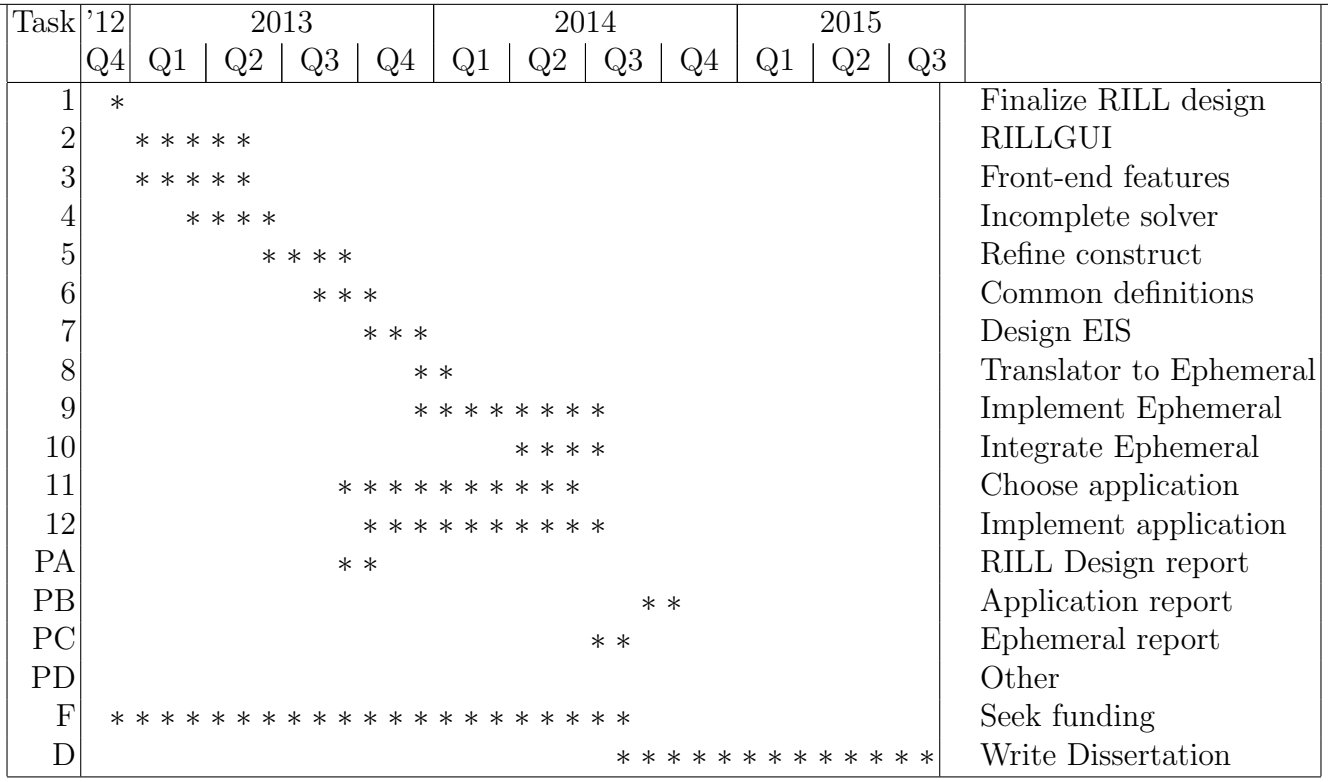

Fig. 1: Schedule (Plan A)

Some of the following additional activities will be necessary:

F: Apply for funding from NSF, or other sources.

D: Write Dissertation.

4.2 "Plan B" Schedule

As considerable time has passed since I anticipated conducting my departmental proposal defense, and the graduate division of UCR insists I finish before 10 years have passed, I recognize that the initial proposal is extremely aggressive considering the time that remains, and I propose the following alternative reduced schedule also be considered. As in Section 4, I first enumerate the (decreased number of) tasks involved in this research, and then propose a specific schedule for their execution.

4.3 Tasks (Plan B)

This alternative decreases the temporal span of this project, by removing tasks associated with Ephemeral, and slightly expands the functionality of the RILL compilation system, to compensate for the lack of a compiled execution platform. The proposed schedule for completion of these tasks is shown in Figure 2.

1. Finalize design of RILL meta-programming features.

   This task is the same as task 1 from plan A (Section 4.1).

2. Construction of RILL-specific editor GUI (RILLGUI) with XML AST tree file interface.

   This task is the same as task 2 from plan A (Section 4.1).

3. Implement front-end features such as manual code manipulations and proof construction/checking.

   This task is the same as task 3 from plan A (Section 4.1).

4. Implement incomplete interpreter/solver.

   This task is the same as task 4 from plan A (Section 4.1).

5. Implement middle-end functions including optimization by RILL Refine constructs.

   This task is the same as task 5 from plan A (Section 4.1).

6. Definition of common mathematical and programming concepts within RILL.

   This task is the same as task 6 from plan A (Section 4.1).

7. Design an alternative to EIS subset of RILL.

   An executable subset of RILL must be explicitly specified and analyzed carefully, to ensure it is efficiently recognizable and efficiently executable by interpreter. Prerequisites: None.

8. Implement recognition algorithm for alternative (to EIS) subset.

   Completion of RILL compilation requires a code that recognizes the alternative subset. Corequisite: Task 7. Prerequisite: Task 4.

9. Solver/Interpreter performance improvements.

   I will improve the interpreter, by allowing the interpreter to recognize performance hints (especially those requiring proofs), so that a larger subset of programs may be efficiently executed. I will also extend the interpreter to make use of multi-threaded parallelism available on distributed shared memory clusters. Prerequisite: Task 4.

10. Choose a simple demonstration application.

    This task is the same as task 11 from plan A (Section 4.1).

11. Implement application in RILL, with translation from requirements to code via domain-specific optimizations, implemented as RILL Refine constructs.

    This task is the same as task 12 from plan A (Section 4.1).

As opportunities arise, it will be advantageous to also perform some of the following publishing-related activities:

PA: This task is unchanged from task PA from plan A (Section 4.1).

PB: This task is unchanged from task PB from plan A (Section 4.1).

PC: Report on the high-performance RILL implementation.

PD: This task is unchanged from task PC from plan A (Section 4.1).

Some of the following additional activities will be necessary:

F: Apply for funding from NSF, or other sources.

D: Write Dissertation.

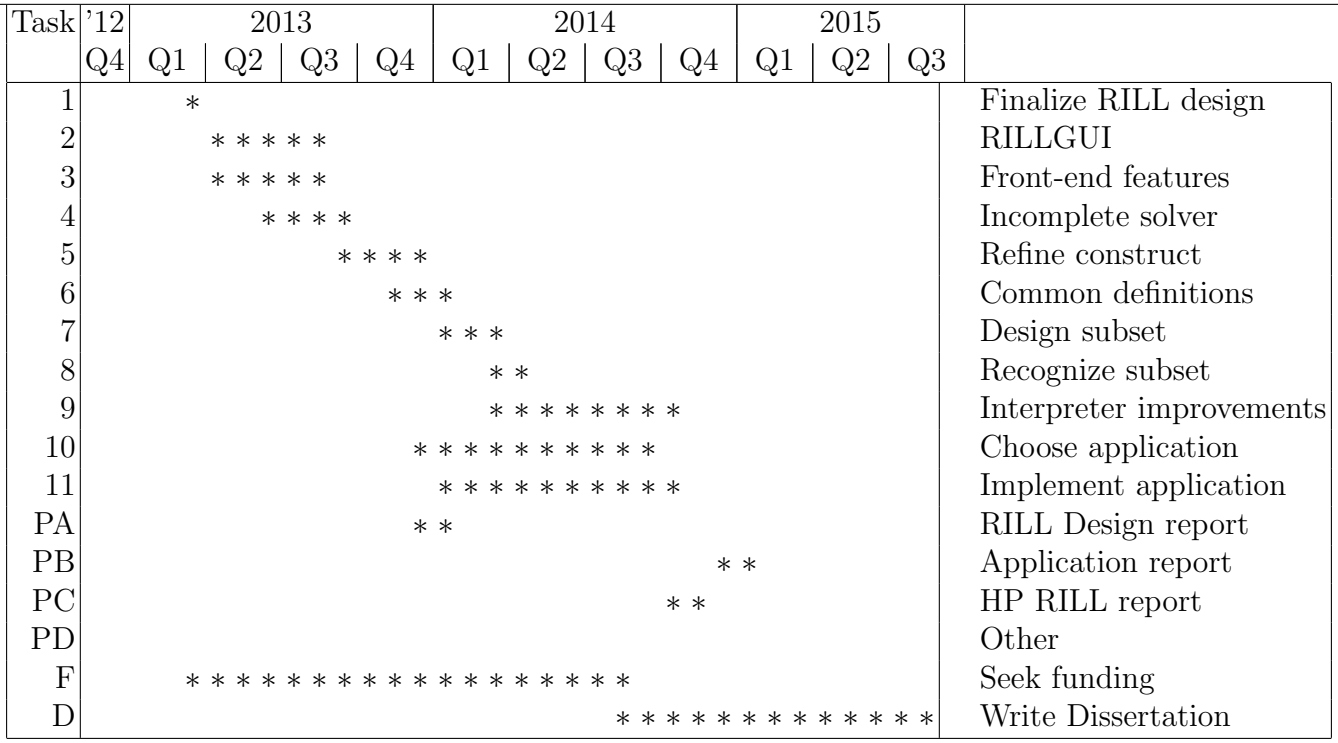

Fig. 2: Schedule (Plan B)

5 Annotated Bibliography

Although this project was chosen 29 years ago, and mostly reached its current form over 20 years ago, this annotated bibliography attempts to relate this proposal to recent literature in computer science. Although this bibliography is intended to situate the present proposal in the context of contemporary literature, the antiquity of the proposal, along with the author's mental accommodation to its eventual completion, makes it easier to evaluate the literature by situating it in the context of the proposal. I considered papers in 44 conference proceedings, drawn from about 300 recent conferences whose titles were evaluated for possible relevance. After reading the abstracts (and frequently much more) of the papers in the 44 chosen conference proceedings, the approximately 44 papers and other publications discussed in this section were selected as being possibly relevant to the current proposal.

The publications fall into several categories:

1. Publications that relate to the theory of the core language RILL, and may provide hints about how to make the last few design decisions, especially those related to meta-programming.

2. Publications that describe programming languages that in some way address similar concerns as RILL or Ephemeral.

3. Publications that provide additional motivation for the present proposal.

4. Publications that illustrate methods that provide scalable parallelism for some applications, that could hint at optimization procedures that could eventually be written within RILL.

5. Finally, publications that turned out to be probably irrelevant.

I have attempted to list the publications in order corresponding to the above list of categories, although some publications fall into more than one of these categories. Each entry references the bibliography at the end of this document for full bibliographic information. Each entry also includes a link, usually to the relevant DOI page, so the reader may easily access the original works. In some cases there is also a link to an on-line copy of the publication itself. In one notable case [§5.36], there is a link to the on-line video of the presentation.

5.1 ARMor: Fully Verified Software Fault Isolation [67]

ARMor is a system that turns any ARM assembly code into trusted code, in the sense that the modified code will not write outside its own data areas, and will not branch outside its own control flow graph (except for returning control and results). The system decorates the code with assertions necessary to guarantee safe behavior (as the paper is mostly about the details of that process, I gloss over a lot here), and proves them using the HOL theorem prover. Guards are inserted before certain statements (indirect stores and branches) if their safety cannot be proven, so that the program, with the guards included, can always be proven safe. Since it is not necessary to insert guards at every risky (indirect) instruction, since some are provably safe, the performance overhead of inserting guard code is usually kept reasonable. It would be possible to decrease the number of places where guard code is required, but at the cost of using a more sophisticated theory within HOL, requiring more ARMor processing time.

This is a nice approach for sandboxing code from questionable sources, as long as plenty of latency is allowed (from the few performance figures they provide (2 cases), ARMor seems to require about about an hour to process each 100 instructions). This is obviously not yet ready for web browser use. Of course all these things will improve.

I would point out the obvious, that simpler theories would suffice if the programmer could provide custom invariants/assertions/proofs for some parts of the code, since the coder should know why/how the code works. This is also desirable, since checking is usually easier than proving, and such invariants/assertions/proofs remain as part of the coder's product, and so may be reused.

From my point of view, this paper is very pleasing, yet also very sad. In 1979 (if not earlier) I had planned to eventually do something (actually more like proof-carrying code than) like this, but was not aware of the availability of HOL or similar systems. In 1983, I finally decided to first construct a decent programming language and proof system, before going about the details of designing a system like this. It is pleasing to see my views have been somewhat vindicated by others. It seems sad that it took so long for others to do this, since HOL has been available since at least the late 80s.

I originally planned to not only enforce the safeness of untrusted code, but also certain temporal properties, such as a limit on the time between Yield statements. This would help to avoid crashes in the operating system I was contemplating.

I have since lost interest in doing that specific assembly-program-oriented project, because my current view is that future massively parallel processors should have a vastly different kind of assembly language, so much so that almost all efforts with extant styles of assembly language will be irrelevant to their coding.

DOI: http://dx.doi.org/10.1145/2038642.2038687

5.2 Specification and Verification of Meta-Programs [6]

Unfortunately, only the abstract of this invited talk is published. The abstract mentions various axes along which Meta-Programming (MP) may be classified:

1. generative - intensional

2. compile time - run time
3. heterogeneous - homogeneous
4. lexical - syntactic
5. reflective - eval-based

The author motivates the talk, pointing out the various forms (compilers, template MP, Java reflection, etc..) of MP that are growing in extant use ("The ubiquity of MP demonstrates its importance"), and pointing out that specification and verification of meta-programs has been only scarcely considered. The remainder of the abstract outlines the talk.

DOI: http://dx.doi.org/10.1145/2103746.2103750

5.3 Nameless, Painless [49]

This paper probably has the most direct practical relevance to my work of all the papers I have read lately. For a program to perform correct transformations on a program, the transformed program must have some "data-like" representation, or encoding. In a language such as RILL, which depends heavily on program transformation, choosing this representation is a very important major component of the language design.

This paper addresses the problem of encoding names and name binders (in the pure functional language Agda) in an elegant and useful maner that does not introduce excessive dependence on the exact representation itself.

Although I have yet to completely understand this paper, it appears they have chosen essentially the same solution as I have. Variable names and binders are accessed through an abstract interface, which can hide any number of concrete implementations. The interface is designed to export just the right level of information, so that it is useful without revealing the underlying representation.

DOI: http://dx.doi.org/10.1145/2034773.2034817

5.4 Program Logics for Homogeneous Meta-Programming [7]

The authors of this paper are apparently the first (other than me) to identify the problem of specification and verification for meta-programming. They address this problem by providing the "first program logic for a generative homogeneous meta-programming language".

It should be clear at this point that I have planned to deal with this problem for some time now, and probably could have published relevant material sooner, had I been allowed to start my research in, say, 2009.

The authors present Pcf_{DP}, a variant of the typed pure functional language CBV PCF, minimally extended for meta-programming. The approach is similar in power to "eval-based" meta-programming, except that misuse (taking eval of garbage) is impossible (The "eval" in this system is not explicitly written). Variables are divided in 2 classes, "modal", and "non-modal". Both have the same type system, but function values in non-modal variables must be functions that are total (guaranteed to terminate). Only non-modal variable names may be applied outside of eval, so that ordinary applications terminate, potentially permitting use of Curry-Howard. Applications within eval may be non-terminating, but those are always buried in quotes, so reasoning outside of eval and quotes remains sound.

The authors then present the logic, typing system, semantics, axioms, and inference rules, in a somewhat abbreviated manner. After several interesting reasoning examples using the system, they go on to prove various theoretical properties, "relatively complete", "observationally complete", and "operationally complete".

http://www.springerlink.com/content/u70826715472r068/

5.5 Directly Reflective Meta-Programming [58]

This paper introduces Archon, a pure functional language intended to support meta-programming in the simplest possible manner. The first 13 pages mostly covers background, with the core material being presented in the next 7 pages. Archon is a variant of lambda calculus that blends lambda calculus semantics with programming language-like semantics, resulting in a fairly simple executable language with well defined semantics and meta-programming features.

Meta-programming of the form that can generate code often involves use of encodings of programs. Directly reflective meta-programming avoids the use of encodings, but the kinds of code that can be generated is usually somewhat limited. Archon is designed to avoid those limitations. Archon code can deconstruct other Archon code, analyze the code, and reconstruct modified Archon code all without using explicit encodings. At present, no type system is given for Archon.

In order to support this, Archon has added three inconveniences:

1. Four primitive functions that are solely involved with code manipulation
2. Two forms of lambda abstraction, one with call-by-name semantics, and the other with call-by-value semantics.
3. Restrictions on reduction order, so that call-by-name parameters may not be reduced until after the call is reduced.

In pure lambda calculus, there would be no significant difference (ignoring name capture issues) between call-by-name and call-by-value semantics, except in the performance of an implementation. In Archon, however, the direct reflectivity depends critically on the distinction, and on the restriction on reduction order.

Even though the semantics is small and well defined, it is much less clear even than most semantics for typed lambda calculi. In most lambda calculi, even those with a very limited reduction order, if some reduction is possible, it is usually harmless to perform that reduction earlier that it would be done in the standard order. Consequently, much speculative parallelism is potentially available to implementations of most lambda calculi. Archon, however has this property severely restricted when meta-terms are used, since they must always be call-by-name parameters. It is important to note that currying is used, and there is no type system, so that it is not always easy to determine if an actual parameter is a call-by-name or a call-by-value parameter.

Most importantly, however; these inconveniences cause difficulty in understanding the meaning of programs.

DOI: http://dx.doi.org/10.1007/s10990-007-9022-0

5.6 Towards Typing for Small-Step Direct Reflection [13]

This paper discusses incremental changes to the set of reflection-related primitives in the, currently untyped, Archon programming language. Desired features for a future typing system for Archon are also discussed. Archon, as previously noted, is a variant of pure lambda calculus, and is intended to support reflective meta-programming without the use of encodings. It does so by adding a few primitives having to do with code inspection and generation, and by adding severe restrictions on reduction order, so that code that "encodes" itself will remain unreduced in cases where it needs to be inspected or needs to be copied for generation of new code.

This version of Archon with the improved set of primitives does appear to be slightly more elegant than the original, and may be more likely to admit some sort of strong typing system. There does not appear to be anything in the changes that ameliorates the problems caused by the restrictions on reduction order, except that the future addition of a type system may permit better analysis of code so that implementations might use a more relaxed reduction order. The semantic complexity remains, so that I would hesitate to consider this form of meta-programming as a part of the proposed work.

DOI: http://dx.doi.org/10.1145/2103746.2103765

5.7 Dependent Types at Work [10]

The Agda programming system (as well as the Coq system) allows a user to write functional programs, with "dependent types" and proofs of some function properties. Properties must be encoded as types, following the Curry-Howard isomorphism. The paper has a readable and somewhat detailed explanation of this use of the Curry-Howard isomorphism. A dependent type parameter of a function has type that depends on the value of some variable name in the scope, such as one of the other function parameters. This way, a function, having the Agda type signature: $(n : Nat) \rightarrow (fishlist : List\{fish\}n) \rightarrow (Listfish(n{+}1))$, takes two parameters, n, and $fishlist$, where n must be a natural number, and the dependent type of $fishlist$ indicates it must be a $List$ of $fish$, of length n. The use of dependent types is necessary to make the Curry-Howard isomorphism easy to use for encoding many function properties. The properties (assertions about values) encoded by types in this way are valid only if the corresponding values are actually produced, so the correctness of such encoded assertions depends on certain functions being total. Conveniently Agda enforces totality of functions.

Agda is reminiscent of a system, I imagined in 1985 but chose not to implement, based on pure lambda calculus and equations of lambda-terms. I was not explicitly aware of the Curry-Howard isomorphism at the time, but was aware of some aspects of it. I had observed the correspondence between conjunctions and product types (which may be encoded in pure lambda calculus). I was also aware of the practical use of application to invoke implication, and considered a system where equations were passed around as encoded lambda terms, so they could be used to provide useful assertions to the compiler. Any equations not involved in the actual output could then be discarded by the compiler, so that their encodings were not manipulated at run-time. I also found a way to obtain the benefits of inductive proofs in this system. Similarly to Agda, equations would have to be produced only by total functions. I chose not to implement this system, or any system based purely on lambda term equations, primarily because of the following two weaknesses.

Although conjunctions of equations may easily be encoded as equations of pairs, in the lambda-theory I chose to use (called $Th(\mathcal{K}^*)$), there is no way to encode disjunctions of equations (or negation, of course). In Agda, disjunctions are encoded as disjoint unions (which in pure lambda calculus could only be encoded as unions with an included determinant (encoded as tuples)). Thus, encoding of disjunctions requires the presence of a procedure which could decide which disjunct is true. This could become a great hindrance when attempting a non-constructive type of proof.

Another weakness is that the universes of pure lambda calculus (even with the liberal definition that I use) and of dependent types contain at most a continuum of entities. This presents a barrier to proofs involving larger types, such as most set theories. In these systems, such proofs could of course be represented by an additional level of encoding, but this would be unnecessarily cumbersome. In practice they utilize UR-elements, or special predefined types, such as "Set", whose actual intended semantics cannot be completely defined within the system, without defining their calculus in terms of set theory in the first place.

A reflective facility has been added to (the only implementation of) Agda, so that the programmer may

write functions (involving certain reflective types) that assist the proof checker (this feature is currently obscure, so this explanation may be approximate). This brings the Agda system close to the current proposal, in that the compiler may run some input code to assist with its processing of other input code.

This is different from the current proposal, since Agda is a purely functional language, where assertions must be encoded as types, whereas language RILL is fully declarative, requiring types and functions to be encoded as assertions on predicates, and as predicates, respectively.

DOI: http://dx.doi.org/10.1007/978-3-642-03153-3_2

5.8 On the bright side of type classes: instance arguments in Agda [25]

This paper turned out not to be significantly related to my research, except in that it lead me to the above paper on Agda.

There are certain observations in this paper about the Agda system which remind me of the way I would like to design such systems. The Agda designers have achieved a nice balance between elegance and pragmatic concerns.

Agda does not have type classes like Haskell has, since the designers decided type classes would add an unnecessary extra level of complexity to understanding programs, the way templates and template meta-programming add complexity to understanding of C++ programs. This paper describes how adding an apparently simple feature (instance arguments) to Agda provides most of the benefits of type classes, while still not introducing excess complexity.

Agda has dependent types, and treats types as first class objects, which may be passed as parameters, and placed in tuples and other structures. Agda also has implicit formal parameters, whose actual parameters need not be written in cases where the compiler will infer their value. This is somewhat like the use of template functions in C++, where the values of type parameters of a function template are inferred from the types of arguments given in the actual function call.

The new feature, instance arguments, is a special form of implicit argument where the compiler does not use the usual implicit argument machinery to infer the actual parameter, but instead looks in the call context for a unique value with the required type. This feature allows the code to use various abstract mathematical structures, merely by importing their declarations, so the compiler can find and apply them when they would be useful.

DOI: http://dx.doi.org/10.1145/2034773.2034796

5.9 Type Checking Modular Multiple Dispatch with Parametric Polymorphism and Multiple Inheritance [2]

The type system underlying the Fortress type system is described, along with the rules contrived to make multiple dispatch compatible with separate compilation. The underlying types include polymorphic object types, polymorphic functions and methods, tuples, and various ground types such as numerics and strings.

The paper gives a set of rules, which are sufficient to guarantee that a specific polymorphic method or function can be chosen at a call site, based on the run-time types of the parameters. It is thereby guaranteed that there exists a single most specific implementation of the polymorphic function/method, given the run-time parameter types. In addition to making the choice of polymorphic function unambiguous, the rules may be checked at module compilation time, since they can be checked locally on the basis of information available during the compilation. In particular, an object type defined in the module being compiled may be extended by an unseen module, but knowledge of the extension is not necessary for rule checking in the original module.

The first three rules given are as follows:

1 : No Duplicates

Two types (of overloaded polymorphic functions/methods with the same name) that extend each other must be the same type

2 : Meet

Two types (of ... with the same name), neither extending the other, requires the existence of a third type (of ... with the same name) extending the first two.

3 : Return Type

Return types of functions must be covariant with input types (of ... with the same name) wrt the extension relation.

Multiple inheritance creates certain problems relating to modularity. Print(String) and Print(Int) may conflict if some type extends both String and Int). To improve that situation, the programmer is allowed to make two types exclusive. So, in the above example, making String and Int exclusive prevents any type from extending both String and Int.

The final rule is:

4 : Multiple Instantiation Exclusion

Different Instantiations of a given type are exclusive, so, for example one object type may not inherit from another in two different ways. Some existing OOP languages (Java) have this rule.

The paper shows that these rules are sufficient to provide the necessary guarantees in a system that allows parametric and ad-hoc polymorphism, multiple inheritance, and multiple symmetric dispatch, while allowing separate compilation to perform necessary analysis.

An algorithm for doing the analysis is given, but the given algorithm (which I did not inspect) is conservatively correct, in that it forbids some legal collections of types and overloaded functions.

My interest in this paper stems entirely from the apparent need to add object types and symmetric multiple dispatch to the Ephemeral low-level language for parallel processing. Ultimately, it was unnecessary to add multiple dispatch (as we know it) to Ephemeral. It still appears necessary to add tuple-like object types with multiple inheritance and possibly "parallel" dispatch to Ephemeral.

DOI: http://dx.doi.org/10.1145/2048066.2048140

5.10 Equational Reasoning about Programs with General Recursion and Call-by-value Semantics [35]

This paper (with 10 authors) describes Sep, a possible core language for the Trellys programming language being designed at 3 major CS schools.

Trellys aims to have the advantages of proving properties of programs via the Curry-Howard correspondence and dependent types, as with Coq and Agda, while leaving behind the dependence on total functions required by those other languages, so that Trellys could support general recursion.

The pure functional language Sep provides these advantages by including features for dealing specifically with non-terminating functions, and by using slightly different rules for proving in the language, as opposed to programming. As one example of this, there is a special type of case statement, which magically distinguishes between terminating and divergent expressions. This can be used in both programs and proofs, but its use in programs must eventually be erased by use of a termination proof for the case expression. Equational reasoning is supported in a way that is effective in this context, in particular, full

normalization is not required, so equivalence between some divergent expressions may be proved. Induction is also supported. A prototype implementation is available online.

In some ways this reminds me of the efforts by Agda implementors to generalize beyond use of functions where termination is easily checked to also allow users to prove termination, thus retaining the benefits of Curry-Howard while allowing a larger class of functions.

Except for the difference between functional programming and predicate logic programming, Sep appears to be the language most similar to the core language I propose. In both cases, a very simple core language is proposed, increasing reliability of proofs about language properties, while leaving programmer conveniences, such as nice syntax, tools, and libraries, to a separate 'surface' language (Trellys) to be implemented as another layer over the core. Both core languages successfully avoid the "Sequential Prison" by merely using a pure language, including no features derived from sequential processing. A primary difference is that predicate logic (as I imagine it) already has adequate expressive machinery for integrating programs and proofs, so that there is no need to use the Curry-Howard correspondence. Consequently my core language is not limited to using a special "terminating" subset of expressions. Hence, no problem remains to be solved, except for efficient implementation.

DOI: http://dx.doi.org/10.1145/2103776.2103780

5.11 How to Make Ad Hoc Proof Automation Less Ad Hoc [27]

In a way, this is the research is the closest to my proposal of any paper I have encountered so far. The authors have succeeded in initiating my agenda in a way that might appear completely unrelated. The Coq system is an interactive theorem prover which also generates programs, via the Curry-Howard isomorphism, from proofs.

In a theorem prover, in addition to the obvious activity of applying lemmas to proven (or given) statements to produce additional proven statements, a variety of other activities are necessary. Many of these activities are somewhat tedious and beg for automation. The paper makes an example of manipulating a statement and a lemma so that the relevance will become obvious enough to the prover so the prover can be made to apply the lemma. In the example, a specific subterm within a term in the statement must be manipulated so that the subterm is moved to a principal position within the term, allowing the prover to recognize that a certain lemma may be applied (after an appropriate substitution).

Current theorem provers, including Coq, provide a facility for writing *tactics* in another language, to automate these tedious tasks. The authors point out that tactics are often brittle, in that they do not easily adapt to new uses. In the above example, the simplest tactic for the situation would commute operators found in specific positions, and so would fail in many slightly different examples. Writing of less brittle tactics requires deeper access to knowledge the prover has, about the terms under consideration, especially about types. The paper then shows how to achieve the desired effect using "Canonical type classes". Their method effectively encodes a form of tactic as a type, that causes Coq's unification machinery to perform the additional transformations in a way that is more adaptable to a variety of situations. Note that much of the above is a serious over-simplification. For accuracy, the paper must be read.

The point is, that objects written within the Coq language itself are used to manipulate, at interactive proving time, Coq proofs in a type-safe manner. The purpose of this manipulation is to make the proving process more convenient for the user.

In my proposal, specifications written in language RILL are used to refine, at compile time (and possibly later), other RILL language specifications in a correctness-preserving manner. Here, the purpose is to improve program performance.

DOI: http://dx.doi.org/10.1145/2034773.2034798

5.12 An Evaluation of Different Modeling Techniques for Iterative Compilation [47]

This is one of the more scientific papers I have come across in this search. Iterative compilation attempts to find a good set of compiler optimization parameters for a program, by comparing the results of multiple compilations, each given different optimization parameters. The space of optimization parameters for serious compilers is usually quite large, so the space must be searched intelligently. Even if the best code was produced when all optimizations are enabled, enabling all optimizations may cause in-feasibly long compile times.

This paper addresses the problem of how to model the problem of choosing a set of optimizations that a compiler should attempt, and compares the effectiveness of three methods, the sequence predictor, the speedup predictor, and the novel tournament predictor. These models are based on the performance of programs as indicated by a collection of run-time counters. These models are first trained using performance counter results from a collection of algorithm kernels, all of which are compiled many times using a fairly large collection of random subsets of compiler optimizations. The models are then judged based on their ability to predict the best collection of optimizations to use on an unseen program, given its performance counter results in the case where it is compiled without optimization.

The sequence predictor is a simple model that takes as input a set of performance counter results, and predicts if a specific optimization should be enabled for the program that produced those results. A prediction model has a sequence predictor for each optimization.

The speedup predictor takes as input a set of performance counter results and a set of compiler optimizations, and predicts the speedup of using those optimizations as compared to using none.

The novel tournament predictor takes as input a set of performance counter results, and two sets of compiler optimizations, and predicts which of the two sets of optimizations will produce the best speedup.

For each type of model, the training was attempted through linear regression, and through support vector machines. All models were trained on the performance counter results for the same large set of kernels, each optimized with the same collection of randomly chosen optimization subsets.

The main result of the paper is that the tournament predictor is superior in wide variety of circumstances. It is also noted that the use of performance counters has been shown to typically provide better performance characterization of a program that does static analysis. I would only note the slight weakness that performance counter results require the ability to run the program, which in turn often requires knowledge of expected inputs. But this weakness is very slight, since much optimization is already somewhat speculatively based on characteristics of the input. The paper itself points out the weakness that it only addresses the problem of which compiler optimizations should be chosen for a given input program, but addresses neither ordering nor repetition.

DOI: http://dx.doi.org/10.1145/2038698.2038711

5.13 The Fortress Language Specification [1]

The Fortress programming language aims to provide ways to do multithreaded and parallel programming while still retaining some of the look of traditional programming languages and accounting for the lessons learned from the last few decades. Fortress provides a relatively traditional data type system, perhaps describable as a cross between Java types and functional types, allowing inheritance between interfaces (traits ala Java interfaces), but without traditional Class types. Most control structures in Fortress are parallel by default, with some allowing the keyword “sequential” to specify sequential behavior. Several syntactic improvements enable the use of relatively math-like notation where appropriate, and include additional opportunities for default parallelism. In particular, Set, Array, and Map comprehensions involve implicit

multithreading. The syntactic enhancements also include additional mathematical operator syntax, but come at some cost, in that many rules are needed to understand exactly how some expressions will be parsed.

A hierarchical system of regions is provided to allow program control of placement of threads, and possibly data structures, although it is not clear how to control the distribution of an array. (Possibly nested) atomic statements are provided so the programmer can avoid race conditions involving shared variables.

Static parameters may be used with functions and Object types, similarly to template parameters in C++.

Contracts allow specification of preconditions, postconditions, and invariants. These must be executable, however.

Overall, ignoring certain enhancements, Fortress is about what I would expect from an effort to extend a 'normal' language, such as Java, to include support for multithreaded series-parallel computation for 'normal' processor systems, such as clusters of x86 multicores. It is an improvement over some previous attempts at parallel languages, and may remain applicable for some time. I can however easily see the possibility of a much better language being implemented as part of the current work.

Of particular interest to me is their specification of Symmetric Multiple Dispatch, the implementation of which is described in a separate paper. This relates to the repair of the Ephemeral language, which may require some sort of multiple dispatch. It is remotely possible that Fortress will make a good target language for an Ephemeral compiler.

http://labs.oracle.com/projects/plrg/fortress.pdf

5.14 The Scala Language Specification Version 2.9 [46]

Scala is a Java-like language with numerous extensions, supporting the development of "domain-specific" languages as libraries within Scala. To this end, parametric (with type parameters) and ad-hoc polymorphic object oriented programming is carefully supported in the language. For example, looping control structures are "virtualized", so that those control structures (say "for" loops) are syntactic sugar for something like iterator calls to the loop variable, so the meaning of loops can be changed by the type of the loop variable. Also, "Implicits" allow some call parameters to be automatically inserted by the compiler, increasing the power of user-defined libraries. An "implicit" parameter, when not given at a call site will be automatically be set to an "implicit" value of the appropriate type if exactly one is visible in the call scope. The actual rules for implicits are more complex, and also allow implicit values to be automatically used as type conversion functions, called "views". The standard library also has some predefined implicits, that allow a user-defined (polymorphic) library to obtain a "manifest" structure describing the type of data as seen by the library user.

There is considerable syntactic flexibility in definition of operator names, allowing library authors to enhance the appearance of code that uses their libraries. Scala also uses the Unicode character set, with its extra operator characters, and allows XML embedding.

The standard implementation compiles to JVM byte codes. That feature made me initially doubt the relevance of this language, since I don't consider the JVM to be a likely platform for massively parallel processing. In fact, there is no mention of parallelism or even threading in the language or the standard library specification, although all the Java libraries are fully accessible.

However, Scala seems to be near the cutting edge of research on hosting application-specific mini languages, which is an alternative approach to some goals of the current proposal. Scala library authors may effectively define (using OOP and meta-programming-like techniques) mini-languages with which to write application code.

Whereas I propose to allow user-defined code transformations in the optimization process for a declarative language, Scala allows users to embed (and optimize in any manner they wish) application-specific languages, allowing the library author and user to guarantee good performance.

The primary weakness I see with this approach, is that Scala does not enforce correctness of such libraries, as they are user-defined and specified, and the language does not provide for their formal specification. Thus, a functional error in the object code could be due to either the user or to the library author. The current proposal requires that code transformations refine entailment, so that functional errors in object code are always traceable to the application specification.

In the arena of dynamically typed languages, it is useful to remember that lisp has long been used to host embedded mini-languages, through the use of the macro mechanism.

http://www.scala-lang.org/node/198

5.15 Scala-Virtualized [44]

Scala-Virtualized is an extension, of the Scala language, that provides improved facilities for hosting domain-specific languages. The three major extensions are as follows: 1. "infix methods" improve syntactic flexibility by effectively allowing the addition of class members externally to the definition of the class. 2. "Fully virtualized" program structures. All program constructs are considered syntactic sugared versions of method calls, rather than just looping constructs as with Scala originally. Consequently, the object system can convert what appears to be a plain Scala program into a strongly typed abstract syntax tree which can then be processed by a user-defined library to produce transformed code. 3. Additional information about source code files is made available to library code, so that user-defined libraries may produce more meaningful error messages when used incorrectly.

Scala-Virtualized has been successfully used to embed SQL queries within Scala programs in a type-safe manner, among other impressive achievements. As Scala-Virtualized is being developed by the developers of Scala, I imagine this represents the next step in the evolution of Scala. Likewise, my comments on the Scala language also apply to Scala-Virtualized. There is no strong support for formal proof of correctness, and no strong support for parallelism or even multi-threading apart from the Java libraries.

DOI: http://dx.doi.org/10.1145/2103746.2103769

5.16 Compilers must speak properties, not just code: CAL: constraint aggregation language for declarative component-coordination [36]

This paper in progress advocates for the use of a declarative constraint language for representing properties of programs deduced by a compiler. The CAL language encodes information about S-Net programs. S-Net is declarative language developed through academia and is being tried in industry. S-Net programs declaratively coordinate stream processing "synchrocells" and "boxes". Boxes are single-input/single-output functional components that may be written in other languages, while synchrocells are written in S-Net and perform stream manipulation tasks, such as pairing, un-pairing, routing et. c. The current plan is to use the aggregated information about an S-Net program, together with some processor information, to feed a constraint solver which will produce a placement solution for the components into cores of a multi-core processor. The developers are optimistic about aggregation of declarative information (functional and performance information) about a whole (parallel) streaming program for purposes such as scheduling multiple cores, and predicting performance.

This is an example of a compilation system, for a (partially) declarative language, that uses a (different) declarative intermediate form for some optimization purpose.

This is also an example of the increasing application of constraint solvers.

DOI: http://dx.doi.org/10.1145/2103736.2103745

5.17 Leveraging Data-Structure Semantics for Efficient Algorithmic Parallelism [20]

This paper describes a novel programmer-assisted method for parallelizing code, based on the understanding that efficient methods for determining some program properties necessary for efficient parallelism cannot be anticipated by the compiler writer. In particular, the data footprint of a computation or of a sub-computation cannot always be represented efficiently as list of memory locations. Also, the footprint cannot always be determined statically, due to the data-dependent nature of many computations, as well as the use of pointers and indexing. The data footprint of computations can be useful for determining which computations (from existing sequentially written code) can be run in parallel. In particular, operations with non-overlapping memory footprint may be executed in parallel. Unfortunately, comparing footprints based on lists of memory locations is infeasible for parallelizing significant programs.

The described system provides C++ templates and run-time components that allow a programmer to provide additional information to enable effective parallelization. The system represents the memory footprint of an operation abstractly, using types supplied by the programmer. The programmer also provides notifications of changes in the memory footprint of an operation, as well as a way of checking overlap (and also checking probability of overlap) in footprints represented by the programmer-supplied types. At run-time, the system calls the programmer-supplied checking functions to determine suitability of allowing parallel execution of the available operations. The system does this in 2 ways. In systems with software transactional memory (STM), this information is used to throttle concurrency based on probability of rollback. For other systems, it is used to obtain guarantees of non-interference before allowing parallel execution.

This can work well, because the programmer often knows an efficient representation of the desired footprint. For example, operations that each perform incremental operations on trees, then recursively descending on subtrees, have a footprint that can be conservatively be represented by reference to the node on which they are currently operating, the footprint being the subtree under that node. In this case, overlap between two footprints can be checked by inspecting the relation between the two referenced nodes.

The disadvantage of this approach (aside from the obvious fact that this implementation only applies to C++) is that there are no correctness guarantees as there are with some systems that rely on static analysis.

This is a very good example of a system that utilizes programmer knowledge (expressed procedurally, in this case) that potentially goes far beyond what can be anticipated by the toolmaker (or compiler writer), to enable greater program performance improvements.

DOI: http://dx.doi.org/10.1145/2016604.2016638

5.18 Efficiently Exploring Compiler Optimization Sequences with Pairwise Pruning [15]

This paper presents interesting work on the "phase ordering problem" for optimizing compilers. An improving progression of three approaches is presented, all based on the justified simplification that the solution can be based approximately on deciding the order separately for each pair of optimization phases. In each approach, some initial measurements are made, for each pair of optimization phases, to determine the best ordering among the two optimizations of the pair (and sometimes numerical estimates of the relative goodness of the two possible orderings), for the program being compiled. Given the pairwise preference information, each approach heuristically orders all the phases in a way that attempts to minimize

some cost measure of the compiled program. When possible, a topological sort is used, otherwise the three approaches differ in the heuristics used. The first approach uses a heuristic that attempts to find and remove a set of pairwise constraints with minimal reduction in a preference metric, so that the remaining constraints admit a topological sort. The first approach also produces the initial pairwise constraints by trying each pair of optimizations on the program, comparing the performance metrics of the programs produced by both orderings of the pair. The results are encouraging, but ultimately this approach suffers from the inaccuracy of applying a pair of optimizations in isolation.

The second approach initially samples a reasonable number of total orderings of the set of optimizations on the program. Goodness metrics are calculated for each pair of optimizations based on the performance metrics for the compiled codes produced for those optimization sequences where the pair occurs. As before, a total ordering is produced based on the pairwise goodness metrics, using a different heuristic. After some refinement, this approach produced very good results. The only disadvantage was that it was still quite expensive to take a sufficient number of sample orderings to produce likely good results.

The final approach takes advantage of an additional observation made on the results of many trials of the previous approach. It was found that typically there are a few dominant pairs, the ordering of which was most important. The effect of ordering of other pairs was usually independent of the orderings of the dominant pairs, whichever choice was made for them. The dominant pairs could usually be found by noting the largest jump in a graph of performance vs sample number when the samples were sorted by performance. Almost always, at least one pair occurred in one order in all samples on the left side of the jump, while being in the opposite order in all samples to the right of the jump.

This feedback approach takes several rounds of sampling. Each round samples some fixed number of sample orders. The main performance gap is found in the performance-sorted sample performance graph. Some decisions are made about pair ordering based on what pair orderings occur to the left and to the right of the performance gap. These decisions are binding on the remaining rounds of sampling, resulting in eventual termination with a total ordering of optimization phases. When no performance gap is found, the remaining ordering decisions are made arbitrarily, terminating the procedure.

In an example, they find a (presumably nearly optimal) optimization sequence of 13 optimizations for a sample program in five rounds, each round using 100 sample sequences. Note most importantly that $5 \times 100 << 13!$.

I would like to guess (conveniently after-the-fact) that the useful independence, of the performance effect of some optimization pair orderings from others, indicates that the phase ordering problem is not exactly the right problem to be solved after all.

I think the above phenomena can be explained as follows: Consider a program AB with two large sections, A, and B. Suppose there are 4 optimizations, p, q, r, and s relevant to these codes. If compiled separately, A has the best performance as long as optimization p is performed before optimization q, while the other optimizations, r, and s, have no effect. On the other hand, B has the best performance whenever phase r occurs before phase s, while p and q have no effect. It is obvious that the performance effect on AB by the choice of order among optimizations p,q will be independent of the performance effect of order among r,s, similarly to the examples studied in the paper. If code A is executed much more frequently than code B, the feedback approach will discover in the first round that p must be before q, and then in the second round, that r must be before s.

Suppose we also have program AC, composed of sections A and C, where A is executed much more frequently than C. Suppose additionally, that C has the best performance when q is before p. When optimizing AC, the feedback method will find in the first round that p must be before q, and the next round will terminate early, finding no performance jumps. The limitations of solutions to the simple phase ordering problem are evident in this example, since it would obviously be better to optimize A and C separately, applying p and then q to A, while applying q and then p to C, rejoining A and C after those

optimizations. Of course, research compilers will also tackle these issues eventually. However, I think that research would benefit greatly from a language (such as the language I propose for this project) that allowed the optimizations and ordering to be specified in the program. Not only will this provide a convenient means for industrial programmers to ensure performance; It will also allow more convenient compiler optimization research.

DOI: http://dx.doi.org/10.1145/2000417.2000421

5.19 Open Language Implementation [62]

This short paper is a plea for more openness and controllability in the language implementation "stack" (compiler and run-time system).

The authors claim the current situation is very awkward for parallel programming, since it is impossible to know what a compiler has done to the code without inspecting the assembly output, impossible to know why the compiler did what it did, since analysis results are not preserved, and impossible to adequately control the compiler's optimizations. Run-time schedulers are also difficult to control, but can have profound unexpected effects on performance of parallel programs.

The authors propose to remedy this situation (for an existing language, X10) through the development of an IDE, integrated with the compiler, that allows the programmer to interactively inspect the intermediate products of the compiler, and view some of the compiler's optimizations as source-source transformations, and choose which transformations should apply to a given section of code, among other things. Beyond that, the IDE would provide additional information useful in syntax/semantic/type debugging as well as run-time debugging, beyond what would normally be provided by a compiler.

This sounds very much like the current proposal, except that it takes the form of an IDE for the object-oriented language X10, rather than a compiler for a pure logic programming language.

My language would instead, when used as a compiler's intermediate language, provide a logical framework where the above could be implemented in a coherent manner, for a variety of programming languages.

When the above IDE is implemented, I think the users will find that much effort is spent interacting with the IDE to select the correct optimizations and sequences to ensure good performance, and that this will become a major component of development effort. This effort could unfortunately be duplicated in similar efforts for other similar programs. To avoid excess duplication of work, the IDE developers will eventually have to create reuseable representations of the information/effort presented to the IDE by the users during interaction. Although some kind of scripting language might seem appropriate for this, I expect that hindsight will show that my current proposal provides the best alternative.

DOI: http://dx.doi.org/10.1145/1984693.1984699

5.20 Distillation with Labelled Transition Systems [30]

The "Distillation" transform for functional programs was introduced in 2007 as an automatic way to perform certain transforms of list processing code previously thought to require mathematical insight. This paper discusses an explanation of how the Distillation transform sometimes produces a genuine algorithmic improvement having super-linear speedup. A correctness proof is also sketched.

This is vaguely reminiscent of a 1984 Lisp conference paper subtitled "Listlessness is better than laziness".

This paper provides yet another example of the many optimization techniques that are somewhat specialized and hence not desirable to incorporate into a general purpose compiler, yet are highly desirable for many programs, necessitating its use in some compilers.

The proposed work solves this dilemma by providing a framework in which a user may safely define and invoke this optimization without modifying the compilation system itself.

DOI: http://dx.doi.org/10.1145/2103746.2103753

5.21 StagedSAC: A Case Study in Performance-Oriented DSL Development. [61]

This paper describes two implementations of SAC (Single Assignment C)-like languages in the Scala-Virtualized framework. SAC is a C-like language with a single-assignment rule for variables, and some additional array manipulation constructs intended for parallel implementation. In the first implementation, "LibrarySAC", a library is defined, utilizing the syntactic flexibility of Scala to provide the programmer with a way of writing SAC-like code within Scala. SAC functionality is then available within Scala through the use of library calls.

The second implementation, "StagedSAC", is accessed through library calls within Scala-Virtualized, but achieves higher performance through code optimizations and a "lightweight modular staging" (LMS) framework, also implemented in Scala-Virtualized. The full virtualization of Scala-Virtualized allows the library code to extract abstract syntax trees of the StagedSAC portion of the code. The library, at compile time, then performs transformations and optimizations on the code, resulting in improved Scala code. A constraint system on array shapes is derived from the program graph, the solution of which sometimes provides partial information on array shapes. Whatever partial information was derivable at compile time about array shapes is then used to optimize array code, via such improvements as removal of redundant index bounds checks, and later, loop specialization. All this information is passed to the LMS framework, which provides common optimizations such as constant folding, CSE, code motion, etc. Other optimizations such as tiling may be applied to improve cache effectiveness. Upon request, the Delite framework may also be used to do additional optimizations and generation of GPU code. One of the major contributions of this work is the ability to use partial shape information in certain ways for optimizing array code.

A results section provides comparisons between various levels of optimization of the StagedSAC implementation (executing on JVM) and a native SAC implementation, for some sample array programs (not using GPU). The fully implemented StagedSAC programs running on JVM had run-times within an order of magnitude of that of their corresponding natively compiled SAC programs.

This illustrates Scala being used for what it does best, the implementation and customized optimization of "Domain-Specific" languages within the bounds of interoperability within a general purpose strongly typed language.

The current proposal suggests the additional step (although along the parallel track of Predicate Logic Programming) of "adding" the advantages of formal correctness proofs for the program and the optimizations applied. I use the word "adding" loosely, since this proposal begins with a language of formal specification, and suggests adding a customizable optimization process. Hence, this paper also provides yet another example of something that could be done better by using the results of the proposed research.

Additionally, it is yet another successful example of the benefit of use of optimizations beyond those innate to the given compiler, illustrating the fact that a given compiler cannot anticipate all needed optimizations.

DOI: http://dx.doi.org/10.1145/2103746.2103762

5.22 Matching Logic: A New Program Verification Approach [54]

This short paper illustrates a new formalism for reasoning about the state of programs, especially sequential programs manipulating structures on the heap, such as is common in C code. I don't understand it clearly, as the notation is not fully explained. It appears to be based on representing the state of the program's data structures as a picture of the structures and pointer relations, much as a programmer would draw on a board, except encoded as a "pattern". The paper apparently shows how to prove (partial) correctness of some simple list processing functions, such as reverse and flatten.

They have implemented this logic in MatchC, a language/compiler (implemented in only "a few months") that can efficiently verify MatchC programs and their pattern specifications.

Apparently this logic is applicable only to systems where the program state explicitly exists, perhaps as a combination of stack and heap structures. I do not wish to limit my intermediate language to such systems. So, this doesn't seem to be actually relevant to my proposal, but it does illustrate how much low-hanging fruit there still is. There are a great many reasonable ideas that are not difficult to implement, yet do not fall into any explored sub-field having its own literature.

DOI: http://dx.doi.org/10.1145/1985793.1985928

5.23 Towards a General Composition Semantics for Rule-Based Model Transformation [64]

This paper attempts to enable compositional use of tools written in multiple languages for model transformations. Module transformation tools written in a single language can be more easily composed, duty to the use of the same type systems, and other language constructs. The paper enables composition of such transformation tools without those advantages, by providing a common virtual machine in which the composition semantics of various model transformation languages may be defined.

This appeared be of interest to my work, because I perceive that it is beneficial to express code transformations and optimizations in a way that minimizes dependence on the actual languages used (for transformed code, as well as transforming code).

Unfortunately, using this work appears unlikely, due to two obvious issues. Most importantly, the definition of the virtual machine and the related language constructs appears to be in itself more complex than anything I would want to use in the core of my system. Secondarily, the virtual machine is a sequential stack-based machine having a few tens of instructions. This is not compatible with the parallel and declarative nature of my intended system. Although such an issue may be repairable, the effort required is likely to be greater than that for complete reconstruction.

http://www.springerlink.com/content/9107303116681321/

5.24 Synthesis of First-Order Dynamic Programming Algorithms [51]

A technique is given for Synthesis (from a declarative and executable specification) of First-Order Dynamic Programming Algorithms via encoding of algorithm testing as a constraint satisfaction problem. Several interesting examples of synthesized algorithms are shown.

At first, it appeared that this paper would have a competitive or complementary technique for my system. After reading, the relevance is not clear. The products (programs) produced by the technique are potentially competitive with hand-coded programs not only in terms of performance, but also in correctness, depending on the programmer. This is because the programs are not certified to be correct. Instead, they pass a test (or a finite set of tests) encoded as a constraint problem.

I find it potentially amusing to consider that the authors have essentially encoded many student's trial-and-error method of programming (when given that the solution is a dynamic programming algorithm).

DOI: http://dx.doi.org/10.1145/2048066.2048076

5.25 Compiling Math to Fast Code [52]

Only the abstract of this paper was available. The talk presumably describes the operation of Spiral, a system for generating efficient code for linear transforms for a variety of platforms. My work cannot benefit significantly from the abstract, as it provides no useful detail.

I would only note that it mentions that compiled code is often orders of magnitude slower than the best code, because the required optimizations are "difficult or impossible" for compilers. The only possible solutions to this involve putting even more optimizations into compilers, thus increasing their complexity, or giving compilers the ability to learn novel optimizations.

DOI: http://dx.doi.org/10.1145/2103746.2103748

5.26 Active Pebbles: Parallel Programming for Data-Driven Applications [66]

Active Pebbles attempts to provide a framework for message-passing parallel programs that naturally utilize small messages with little or no coherence or predictability in the message flow patterns. Such programs may experience performance problems on traditional platforms, due to their high per-message overhead for inter-processor communications. Pebbles are (potentially very small) messages sent between entities. There may be very many senders, and very many receivers, called targets (also potentially very small). One may imagine the data flow pattern as a storm in a room full of tiny pebbles. There is no expectation of regular patterns or coherence. They provide results showing that Active Pebbles gives good performance for various benchmarks on top of various parallel programming platforms. The authors list 5 mechanisms responsible for the performance of their framework.

1 Fine-Grained Pebble Addressing: Pebbles are addressed directly to their target, with an address of reasonable size.

2 Message Coalescing: Pebbles may be aggregated temporarily into larger messages by the AP mechanisms, to decrease the messaging overhead on platforms that only efficiently support larger messages.

3 Active Routing: Pebble flow is slightly adjusted, to increase the probability of message coalescing.

4 Message Reduction: Depending on program semantics, certain pebbles may be reduced in transit, if they are coalesced into the same message.

5 Termination Detection: Support is provided for some distributed quiescence detection algorithms.

I find 1, 2, and 3 most interesting, as these are the mechanisms that overcome the small-message penalty on some platforms. This occurs with the penalty of small additional latencies introduced by the active routing and coalescence mechanisms. Fortunately, these mechanisms are programmable, and can be adjusted to also efficiently handle workloads with well-understood communication patterns. The observed performance justifies my assumption (in Ephemeral) that small messages (within the computing/programming model) are reasonably efficient and sufficient for all parallel programming needs.

This may provide a convenient implementation target for the parallel compiler, if there is not sufficient time to implement Ephemeral. If there is time to implement Ephemeral, it would be wise to consider use of some mechanisms from Active pebbles.

DOI: http://dx.doi.org/10.1145/1995896.1995934

5.27 Software Synthesis Procedures [37]

The authors of this paper have made some of the same observations I have (Section 1.1.1), and also utilize them for programming language improvement. They have enhanced (presumably via a library) an existing sequential imperative language (Scala) to include features that calculate solutions to declarative formulae. The features are convenient for establishing preconditions corresponding to the required formulae. The declarative formulae must belong to some class chosen by the developers. Reminiscent of Section 1.1.1, the developers have chosen several classes of formulae, and plan to include additional classes. The classes of formulae have additional constraints, one of which is that the class of formulae must be decidable and must have a decision procedure that also produces witnesses, thus, this approach is limited to a subset of the classes described in Section 1.1.1. The additional constraints provide the advantage that the compiler can always determine if there is a possibility for the computation to fail through lack of solutions, or may be nondeterministic due to excess solutions. The decision procedure is converted (potentially automatically, though now only manually) into a 'synthesis procedure' which, given a formula in the relevant class, produces code that efficiently solves the formula for the values of any needed variables, or determines if a solution exists.

Novel language features are described that provide access to the declarative solution functionality.

The expression: choose($(x : Int, y : Int) => 5 * x + 7 * y = s \wedge x \geq 0 \wedge y \geq 0$) (assuming $s \geq 0$) returns a pair (x', y') such that $5 * x' + 7 * y' = s$, for any value that s has at runtime. It will also produce a compile time warning that solutions will not be unique for some values of s.

The expression: given $x : Int => y + x = 3 * k \wedge x > 0$ have $2 * x$ else 0 returns $6 * k - 2 * y$ when $3 * k > y$, otherwise it returns 0

The paper describes the use of formulae in the two classes implemented so far. These are:

1. Linear equations and inequalities where coefficients may be unknown until runtime.

2. Simple formulae involving finite sets with linear constraints on set sizes.

I see this as very incremental, compared to the proposed project, as it represents a very cautious use of declarative programming, limited to certain classes of decidable formulae, and applies this as a minor extension to an extant sequential imperative language. The proposed project, however, allows the programmer to apply arbitrary correct program refinements to declarative code, allowing it to be implemented efficiently. It should also be obvious that the current proposal would provide a convenient framework in which the work of this paper could be well supported and easily generalized.

DOI: http://dx.doi.org/10.1145/2076450.2076472

5.28 Controlling Loops in Parallel Mercury Code [9]

Mercury is a logic programming language similar to Prolog. It has similar syntax and semantics. It also has a specific execution model upon which programmers can depend. Unlike prolog, it also has input/output modes for predicate parameters, a parallel conjunction with semantics similar to sequential conjunction, and partial nondeterminism limits associated with parallelism (parallel conjuncts are required to be deterministic). The compiler and run-time system internally use futures for parallel communication, and an "engine" construct for processor utilization control, and "context" concept, relating stacks to threads.

It appears that the paper solves a problem that actually resulted from the techniques used in the current Mercury implementation. Stacks are used as in many other LPLs to control backtracking, except here, each concurrent goal (conjunction) of a parallel conjunction is given its own stack. For many programs this works well, and produces nearly optimal speedup on closely coupled shared-memory SMP hardware. In

the case of tail-recursive programs that include parallel conjuncts, the memory performance is poor, due to the need to keep many stacks. This happened primarily in cases where the tail-recursion occurred within a parallel conjunct (often simply converted/parallelized from a loop construct). The current implementation ran a loop iteration from the "current" stack, and spawned another thread/context/stack to run the remaining iterations of the loop. This is because the first parallel conjunct is executed on the current stack, while other conjuncts are executed on spawned stacks, and tail-recursive calls are usually placed in the later conjuncts. Since the current stack contains the loop control, it cannot be discarded after the first iteration, as one might naively expect.

This often resulting in using n stacks for a loop with n iterations, for loops produced from map_left/right calls, since control had to be passed back to previous iterations, requiring stack retention even in the presence of apparent tail recursion.

The paper describes how the problem is solved by additional analysis and code manipulation in great detail, along with ideas on how their method may be generalized to handle even more cases.

Their solution involves adding to the run-time a loop control data structure, used to manage all the threads and contexts associated with a particular looping construct. Additional analysis is performed to ensure that tail-recursive looping constructs are properly identified, and that a limited number (related to the available number of "engines") of parallel iterations/stacks are sufficient for loop execution.

I think this paper is in some way especially un-applicable to my research, in that it relates to an impure (therefore not fully) declarative language, where a quite complex optimization technique is being added into the compiler to handle a fairly common case.

I would predict future work will reveal the continuing need to add more complex optimizations to the same compiler/run-time system, eventually resulting in serious maintenance difficulty and bloat.

If anything this paper illustrates the need for a compilation system that allows the set of optimization techniques to be expanded externally, without expanding the compilation system itself, as I propose for my dissertation.

DOI: http://dx.doi.org/10.1145/2103736.2103739

5.29 Heterogeneous Actor Modeling [39]

This paper describes the heterogeneous capabilities of the Ptolemy modeling system. The system defines a generic actor framework in which various system composition methods may be defined. These methods are termed Models of Computation (MoCs). The idea is that systems and their components may be described as actors (related to the actor models of Hewitt), and that different MoCs (such as data flow, discrete event, FA, etc) can be described as different ways of coordinating the processing and communication among actors representing the components of a system.

The details (not given here) of their specific actor model are designed to facilitate this. Component actors may be composed into larger actors or systems, and a special entity, called a coordinator, build into each composition, controls how the components interact.

Each MoC is implemented as a special coordinator. The paper briefly describes each of 6 MoCs implemented so far (process networks, dataflow, discrete events, finite state machines, continuous time, and rendezvous), and how hybrid systems may be easily described heterogeneously using these compositional MoCs.

This work is not directly related to my declarative language project, but it is interesting to compare and contrast with my low-level parallel computing model (Ephemeral). The Ptolemy system is an actual actor-based system, while Ephemeral is actor-based only in the sense that it is even less like anything else than like actors. All actor models, that I have seen, allow actors to persist indefinitely and to have local state, while in Ephemeral, "places" are ephemeral in nature, and their duration is always finite and may

be statically determined. The activity that occurs in a "place" is effectively atomic, so that the entire state of a system is present in the messages. As in actors, Ephemeral communications occur by messages, although Ephemeral messages are individually "small" in that their size may be statically determined, while some actor systems allow arbitrarily complex data structures in messages. I feel that the most important difference is that actor-based systems are inherently polluted by carrying the sequential model of computing, in that actors are very thread-like, having indefinite duration and access to a (possibly large) local store or collection of variables. I imagine this occurred so that actor model simulations could be carried out efficiently on the stock hardware of the 70's. Unfortunately, this means that the same type of hardware will remain the natural execution platform for actor-based computing. Although many actors may be composed to utilize a massively parallel platform, the style of code implementing the individual actors requires that the individual processors be somewhat traditional. This means that processors with traditional control mechanisms and large memory are required, potentially stalling progress toward the use of larger numbers of much simpler processors. Ephemeral does not have this problem, and promotes the use of small simple processors.

DOI: http://dx.doi.org/10.1145/2038642.2038646

5.30 An Automatic Parallelization Framework for Algebraic Computation Systems [40]

This system injects map-reduce parallelism into exact-arithmetic programs. The main input to the system is an imperative program in AXIOM, a conventional control-flow oriented sequential language (with some facility for expressing parallelism) oriented to computer algebra applications. Using static analysis, user hints, and descriptions of algebraic properties of some functions, also provided by the user, the system detects 'accumulation loops', and verifies the associativity of the operators involved. It then reconfigures those loops as parallel map-reduce code.

This system, and the user's property descriptions are written in a separate language, Spad.

This system would be in some sense a subset of my system, if all of the above languages were the same declarative language. My system provides a framework in which systems such as this one may be conveniently implemented and reused. Conveniently, this system illustrates a class of optimizations one might wish to implement using my system. This is rather an approximation, since these optimizations, as expressed in this system, are bound to the control flow graph input representation of the input language, and so is only directly applicable to imperative programs. It is however imaginable that some variation of these optimizations are applicable to some declarative programs, and so it is possible that the core ideas in this work could be beneficially reused within the framework of my system.

DOI: http://dx.doi.org/10.1145/1993886.1993923

5.31 Position paper: Using a "Codelet" Program Execution Model for Exascale Machines. [68]

The authors observe that course-grained parallelism, of the type that is efficiently supported by extant architectures, denies the run-time system of certain opportunities for adaptation, and tends to require large overheads for certain operations, such as task swapping and migration. The authors claim that their work indicates improvements are possible by dividing the program into smaller pieces (codelets) for execution.

The Codelet model they propose appears to be a hybrid between Ephemeral and the dataflow model.

It is difficult for me to see this work as original, as I have seen many hybrid dataflow systems proposed since the early 1980's, and the dataflow model is certainly no stranger to the use of small executable units. The part that seems novel to me is the claim that their work supports this. It's too bad the paper gives no

details about, or references to such work. But I suppose that is to be expected from a paper that mentions IBM Cyclops-64, and Intel's single chip cloud machine, but not Sun's Niagara.

A generic hardware model is also mentioned, having a hierarchy of 4 levels (system, node, chip, and cluster), the typical unit for each level containing an interconnect and multiple units of a lower level attached to the interconnect. The node level unit also has extra DRAM banks attached to its interconnect. The lowest, cluster, level has computing units (CU) and at least one scheduling unit (SU), and a cluster memory, and interconnect between them. CUs and SUs have local memory and multiple register sets. SUs also have the ability to communicate with SUs in other clusters, presumably for load balancing and migration.

The relation between the hardware model and the codelet model is not clearly explained. My guess is that this paper was trimmed from a larger paper, and the hardware model was left in to provide some notion of what kind of computer would benefit from use of the codelet model.

DOI: http://dx.doi.org/10.1145/2000417.2000424

5.32 Adaptive Runtime Selection of Parallel Schedules in the Polytope Model [50]

This is limited to Polytope model computations, but has improved accuracy compared to many other techniques. The limitation to possibly parametric polytope model computations allows analytical determination of loop iteration counts and array sizes after the input array sizes are known. This, in turn, allows improved performance prediction.

The compiler may produce multiple versions of the code depending on how flexible the array data dependencies are. Their method produces a modified code at compile time which performs profiling on itself, along with the production code for performance execution. At install time, each code version is profiled systematically, with various numbers of threads, various input array sizes/shapes, and various tile sizes. The profile data, after tabulation, effectively defines a performance model for the code. Finally, at run time, the input sizes are known, and the number of available processors/threads is known approximately, so that the model can be used to predict the performance of the best variation of each version of the code. For each version, the best parameters can be predicted, such as number of threads to use, and tile sizes, depending on input sizing and processor loading, etc.

This method performs quite well at selecting a good version of the code to execute, because it is able to account for almost all factors influencing performance. These include: processor design (considered by load-time profiling), Input data size/shape (accounted for by systematic profiling using Polytope model) Processor resource availability/loading (via profiling with various numbers of threads)

What I find relevant about this work is that it explores one of many instances where a large class of computational problems (HPC codes) which are in general difficult to optimize, has a frequently occurring subset of problems (in this case, problems expressible as coherent loop nests with simple dependencies) which, as this research now shows, are relatively easy to optimize well, even for parallel processing.

My proposed project will provide a framework in which such cases may be defined and recognized, and in which to express what needs to be done in such cases, without requiring actual compiler modification.

http://dl.acm.org/citation.cfm?id=2048588

5.33 The Elephant and the Mice: The Role of Non-Strict Fine-Grained Synchronization for Modern Many-Core Architectures [53]

This paper explores 4 questions, among them is the performance gain achievable by the use of non-strict fine-grained synchronization (dataflow tokens, full/empty tag bits, synchronization state buffer), as

compared with other synchronization mechanisms (barriers, signal-wait).

The authors implemented three fine-grained synchronization mechanisms for the (single-chip) IBM Cyclops-64, and tested it in very carefully crafted simulations to ensure accuracy of results. The IBM Cyclops-64 has a relatively symmetric NUMA architecture with a large crossbar and 160 "thread units" (single issue cores) in pairs having a shared FPU and a crossbar port. Each group of 10 thread units also share an instruction cache, every four of which also share a crossbar port. Each core has its own data memory, and direct (but higher latency) access to all other cores data memories through the crossbar. There are no data caches. The cores have scoreboarding which allows some out-of-order execution and write back. There is also a common high-speed synchronization bus shared by all cores.

The mechanism(s) implemented were integrated deeply into the architecture in the form of an Extended Synchronization State Buffer (ESSB). The ESSB essentially simulates the use of imaginary tag bits attached to some memory locations chosen implicitly by the program. Special load and store instructions utilize the ESSB. In the most effective mechanism tried, ESSB3, a special store sets the full bit associated with the memory location, after which the special load instruction resets it. If the load instruction occurs before the needed value has been stored to the location, a stall may eventually occur, but the thread will be automatically resumed once the data becomes available. The load instruction still issues but the thread will not stall until the value itself is needed by an operation.

The ESSB mechanisms, along with barriers and signal-wait, were all tested with customized versions of a number of benchmarks. As hoped for, the ESSB3 versions of all programs scaled well as long as there was available parallelism, while performance of the equivalent versions using other mechanisms became limited due to synchronization overhead. In particular, a case of the wavefront benchmark gave almost linear speed up out to 160 threads with ESSB3, while the best of the other methods (signal-wait) only scaled to about 115 threads.

One of the other results of the paper is that introduction of fine-grained synchronization increases the hardware size by at most 10%, that almost entirely due to the ESSB cache-like structure itself.

I claim this provides additional justification for doing things in a fine-grained manner in parallel computing models, as in Ephemeral, although this paper only addresses synchronization methods.

DOI: http://dx.doi.org/10.1145/1995896.1995948

5.34 Programmable Data dependencies and Placements [12]

This paper is based on the factoring of a (data-independent) program into a data dependency graph, and the individual computations performed at graph nodes. The proposed use of a data dependency algebra (DDA) to generate the dependency graph is described, as well as use of a space-time DDA (STA) to describe the communication topology of a parallel processor architecture. The main idea is that a (data-independent) program can be expessed as the following three modules:

1. The algorithm, independent of data dependency.
2. The data dependency graph expressed as a DDA.
3. The mapping/embedding from the DDA to a the STA of the execution platform.

and that 3 can be generated automatically from 2 and the platform STA, while the programmer factors the program into 1 and 2. The compiler writers are to be responsible for generating the STA graph descriptions of target platforms. The paper goes on to describe DDAs for various parallel algorithms, then STAs for various processing topologies, then various embeddings of DDAs into STAs, and then code generation. The system was not yet implemented, so the performance results are due to manually "compiled" code.

This system is simillar to a proprietary system "GAUSS" that I 'almost' implemented in 1989. GAUSS did not require programs to be data-independent, however data-dependent programs required more programmer annotation (manual allocation of processor resources).

It is also simillar to another proprietary system I proposed later in 1989, that would automatically handle C code, limited to the data-independent case.

This paper, together with some of its referrences, shows that automatic placement of computations onto processors in various topologies has long advanced to a point sufficient to support many practical high performance applications.

DOI: http://dx.doi.org/10.1145/2103736.2103741

5.35 Expressive array constructs in an embedded GPU kernel programming language [19]

This paper describes the experimental addition of a secondary kind of array to the Obsidian GPU Kernel Programming Language.

Obsidian is a Haskell package for generating GPU kernels. Using Obsidian, one writes Haskell code that produces a Kernel. Unlike many code generators, the Obsidian/Haskell program resembles the generated kernel, so the programming process is more like writing a kernel in a high level language than like writing a code generator. Obsidian provides GPU-limited versions of the usual programming constructs, including arrays. The existing array construct, "pull" arrays, may be read using complex index expressions, but may only be constructed (written) using coherent indexing, such as an affine function of the threadid. This works quite well sometimes, for example, allowing a simple form of automatic loop fusion. In some cases, such as array concatenation, this is inefficient, as it requires use of conditionals within a kernel. The situation is improved through the use of the novel "push" arrays. Push arrays may be written using complex indexing expressions, but may not be read that way. After construction, a push array may then be effectively converted to a pull array, requiring insertion of synchronization code between these different uses. The paper proceeds to illustrate the use of push arrays via simple parallel sorting kernels (Batcher's bitonic algorithm) and performance measurements.

This is another example of a case where a particular optimization (separation of gather and scatter operations into layers with intervening synchronization), has the following characteristics: One would not expect it to be built into any compiler, but research shows it is quite effective for certain specialized situations. It seems obvious enough to be discoverable by any serious programmer.

DOI: http://dx.doi.org/10.1145/2103736.2103740

5.36 The Sequential Prison [59]

The computer graphics (and OOP) pioneer, Ivan Sutherland, makes the case that the self-propagating cycle of learning and teaching sequential programming has become so entrenched that there will be no escape without a significant change in the way computation is seen. More specifically, he claims that even the use of languages encoded as sequences of characters causes a sequential bias in our thinking about computations so expressed. Many other observations are made in his talk. Asynchronous logic is almost never used in modern systems, even though it potentially offers considerable energy savings. This may be evidence that computer engineers are stuck in a rut (sequential clocked vs. self-timed logic) similar to that (sequential vs. parallel) of computer scientists. The way programming is described contributes to the sequential prison. A program is usually defined as a sequence of steps. Sutherland says that computer scientists need to stop using the word "programming" for what they do, and use another word, such as (perhaps) "configuration", if they are to ever escape the rut.

The current proposal seeks to avoid becoming trapped in the sequential prison, in various ways including the following: Predicate logic programming is used at all levels, so that effort is required to introduce sequencing. Only an abstract syntax is defined, to avoid the necessity of expressing programs as a sequence of bytes. The system target is the Ephemeral language, which itself avoids sequential bias.

DOI: http://dx.doi.org/10.1145/2048066.2048068
VIDEO: http://dl.acm.org/ft_gateway.cfm?id=2048068&ftid=1163270
&dwn=1&CFID=120492121&CFTOKEN=31776533

5.37 Adapt or become extinct! The Case for a Unified Framework for Deployment-Time Optimization [28]

I should perhaps take this advice, as this project was hatched in the early 80s. =)

This paper advocates the adoption of adaptation in many forms as a necessity for future high - performance computing applications, especially for scaling the "walls" of memory, communications, parallel programming, power, etc.

It appears that homogeneity may never arrive in the area of high-performance/high-efficiency computing the way it has for desktop computing. This in mind, the authors point out that a code that is tuned for one system is quite likely to perform poorly on another system, due to potential differences in a variety of system attributes, all of which may need to be accounted for in the code if it is to perform well. These attributes include ISA, number of processors, memory per processor, interconnection network, cache sharing and cache hierarchy, among others.

Various extant adaptation strategies are discussed, such as optimizing compilers, algorithms with various adjustable performance parameters, auto-tuning libraries, choice of different communication packages, scheduling methods, and cache-coherence protocols.

It is also pointed out that sometimes adaptation must be done at run-time, for example choosing matrix representations and algorithms based on the sparseness of data and on the nature of sub-structures. Another example is the use of schedulers that schedule complementary workloads together to optimize resource utilization. Another example is use of profiling data.

It is also pointed out that all these techniques are applied in a rather ad-hoc manner, and that a more holistic approach will be necessary in the future. The authors propose an "adaptation infrastructure" where a single decision maker receives information both at run-time, and before, in the form of programmer annotations, static analysis, profile data etc., and uses all this information to make adaptation decisions of all kinds, both before and during run-time.

I would only point out that having a programming language capable of expressing and controlling compiler optimizations could greatly simplify the task of creating such an infrastructure, especially in the case where correctness is required.

DOI: http://dx.doi.org/10.1145/2000417.2000422

5.38 Implementation of a Hierarchical N-Body Simulator Using The OmpSs Programming Model [48]

This paper describes lessons learned parallelizing Treecode, an N-body gravitation simulator using the Barns-Hut algorithm, for execution on a moderately (4 6-core Xeon with a total of 48 GB) parallel system.

The OmpSs system is an extension of OpenMP, with additional annotations to designate data flow directionality between tasks. OmpSs also adds an extra thread for each processor to manage data-flow-like task synchronization.

The authors consider this algorithm to be a representative example of "irregular scientific applications", in that it solves a problem that could be solved using array computing on a regular grid, but requires less computation, by doing things in a data-dependent manner.

Even on this system with large traditional processors having large memories, the main lesson reported is that finer grained tasks were much better for load balancing, and that processors should include support for task creation and scheduling for larger numbers of smaller tasks to improve processor utilization.

DOI: http://dx.doi.org/10.1145/2089142.2089150

5.39 Two for the Price of One: A Model for Parallel and Incremental Computation [11]

When VisiCalc first came out, I did a bit of thinking about incremental computation. When I was hired in 1980 to work on an incremental assembler, I also thought about higher forms of incremental computation, and eventually concluded that some things (easy split-hard join / parallel prefix) that benefit parallel computation also benefit incremental computation, as do the authors of this paper.

At first it appeared that this paper might have some unusual insights about parallel (and incremental) computing and optimization. It turns out to be yet another paper describing the performance of programs written with certain restrictions and added features, such that it could be optimized using certain techniques.

My system could provide a context in which the techniques of this paper could be applied in an organized manner in conjunction with other techniques. It implicitly identifies a class of programs to which certain optimizations may be applied. There is therefore a chance that this paper may provide useful examples on which to apply the proposed work.

The paper applies the parallel programming techniques of concurrent revisions with isolation types to the task of incremental computation. They do so without losing parallelism. This comes at some expense as programs must be altered to conform to this scheme. Performance results seem to show that the expense is justified, since many of their programs appear to have speedup better than that attained through parallelism alone.

DOI: http://dx.doi.org/10.1145/2048066.2048101

5.40 Theorem-based Circuit Derivation in Cryptol [38]

Apparently, only the abstract is published. I think the most important part of the abstract is the first paragraph, especially the words: "In theory, transforming a high-level specification into a high-performance implementation is an ideal means of producing a correct design, it is hard to make it work ... . This talk describes an exception."

I hope my dissertation will describe another exception, with a broader scope.

It should be noted that Cryptol is a functional language (almost exactly Haskell) for generating circuit descriptions.

This reminds me of bluespec.com which also has a Haskell-based system for circuit synthesis, described in a past UCR CS colloquium.

DOI: http://dx.doi.org/10.1145/2047862.2047894

5.41 Java Dust: How Small Can Embedded Java Be? [14]

This paper briefly describes Muvium, a Java compiler that targets embedded microcontrolers implemented on FPGAs (and without an operating system). It's targets, Leros (16 bit), and PIC (8 bit), are also very

briefly described. A special compiler optimization for this situation is also briefly described, along with the compiler.

Muvium supports only a small subset of the JDK appropriate for deeply embedded applications, but adds some appropriate extensions. Integers are stored as 16-bit values. The processor circuitry (not including memory) of Leros can be implemented in 235 logic cells on contemporary FPGAs. A very simple program together with a Leros processor and a serial port extension occupies a total of 435 logic cells (104 of which were the program rom) and a memory block for the RAM. Usable experimental embedded applications have also been implemented this way using a few hundred logic cells, and a single RAM block with $<$ 1K bytes RAM, and a few K bytes program rom.

This appears to be the smallest footprint implementation yet of Java programs.

The main point here is that the smallest current implementation of a message passing language is done in such a way that it still requires a few K bytes of memory per processor for a meaningful application. This is in contrast to my approach which targets processors having only a few bytes of message buffers and no RAM storage.

DOI: http://dx.doi.org/10.1145/2043910.2043931

5.42 Hybrid Partial Evaluation [56]

This paper introduces 'hybrid' partial evaluation via the toy language MOOL, resembling Java. Like on-line partial evaluators, extensive binding-time analysis is not performed, but like off-line partial evaluators, all partial evaluation transformations are done at compile time. In leu of analysis, programmer annotations are required to indicate expressions which should be evaluated at compile time. This apparent burden is considerably eased by policies that automatically evaluate some things whose evaluation at compile time becomes possible due to other expressions being annotated by the programmer. Civet is a 'hybrid' partial evaluator for Java, developed according to the same semantics described for MOOL hybrid partial evaluation. Civet was developed in 4 person-months as a modification of an existing compiler. The paper also compares performance of programs processed by Civet with performance of programs processed by Jspec, an existing Java specializer, with Civet usually providing faster run-times with minimal additional programmer effort.

Although this paper turned out to be less relevant than I expected, What I find most interesting about it is that, over a decade after research in partial evaluation had its own conference (PEPM), in practice, a minimal effort in manual annotation still provides results superior to the automatic approach.

I think this only adds to the motivational case for my research.

It is also interesting to note that this paper could have just as easily been published 20 years ago, as there is no significant dependence on more recent technology or research, other than, possibly, motivational dependence. The researchers simply took a path which was always feasible, but had been long overlooked, as I am taking also doing in my research.

DOI: http://dx.doi.org/10.1145/2048066.2048098

5.43 A Step Towards Transparent Integration of input-Consciousness into Dynamic Program Optimizations [60]

Not actually relevant. This is a competing approach that seems to require a large number of runs to learn input-sensitivity related things about the program automatically and apply that information to optimization.

DOI: http://dx.doi.org/10.1145/2048066.2048103

5.44 Model-driven engineering and optimizing compilers: a bridge too far? [26]

A random sampling of paragraphs from this paper seemed to indicate that the authors were doing almost the same project as I was. It was not actually the case, but much of what they recommend is likely to find a place in my work, and my work may find a place in what they recommend. The main techniques promoted here are use of source-source code transformation MDE tools available in IDEs to prototype compiler optimizations, as well as taking full advantage of MDE/IDE tools generally to improve compiler code maintenance.

They report good experiences with their attempts to use MDE techniques in program development, application of compiler optimizations, and compiler development, in connection with a couple of language development projects.

http://www.springerlink.com/content/h286827658222334/

5.45 Reasoning about Metamodeling with Formal Specifications and Automatic Proofs [34]

This paper superficially appears potentially closely related to my proposal, as it describes methods for manipulating models and meta-models involving model transformation, and formal proofs are somehow involved. A closer inspection reveals a number of confounding factors making it difficult to determine the relevance of this paper. The models/metamodels involved are of the UML variety, and the model-metamodel relationship used here is not exactly what I envision using in my proposal. Also, the language ("FORMULA") they use seems to contain a number of elements that seem somewhat arbitrary, possibly chosen for the convenience of some particular research. I may examine this work more closely, as well as some of the references, to determine if there is anything I can reuse.

http://www.springerlink.com/content/37q86r883j03m3u6/

A Background Topics

A.1 Lambda Calculus

The Lambda Calculus [4], originally devised by Alonzo Church [18], is (nearly, if not actually) the simplest system capable of expressing all computable functions. Lambda calculus has been used as a computational model for many functional programming languages, and is often used in the definition of semantics of non-functional programming languages. Lambda calculus forms the basis of the majority of competitive efforts involving formal verification of correctness of programs, to which the proposed project is related. Concepts from lambda calculus are used extensively in my description of *Lambda Predicate Logic* (see Section A.2), which provides the foundation for the proposed RILL language.

I will actually describe a version of lambda calculus consequent to a particular theory ($Th(\mathcal{K}^*)$) [4], which I will simply refer to as Lambda Calculus. The following description will be less detailed than the usual formal description of lambda calculus, since I do not wish to commit to the notion of Lambda Calculus as a language of finite strings. This form of Lambda Calculus works perfectly well as a language of finite or infinite trees.

Lambda Calculus provides a simple way to write intensional function definitions as follows:

$\lambda x.body$

denotes a function of a single formal parameter x, where the *body* expression may have free occurrences of the variable x, referring to the formal parameter x. The result of applying this function to an actual parameter X is the result of evaluating the expression *body*, after free occurrences of the variable x have been replaced with the value of the parameter X. By way of a loose example, the function $\lambda x.(x^2 + 3x)$, applied to the actual parameter $y + 1$, evaluates to the same value as the expression $(y + 1)^2 + 3(y + 1)$, which expression is constructed by replacing free occurrences of the variable x in the expression $x^2 + 3x$, with the actual parameter $y+1$. "Pure" Lambda Calculus actually does not admit ordinary mathematical notation, as was used in the above example, so I will refrain from additional looseness. The only operator in Lambda Calculus is application of a function to an argument (the actual parameter), and is denoted by juxtaposition of expressions.

A simplified abstract syntax for Lambda Calculus is:

1 $E ::= (E_1 E_2)$ ← application of function E_1 to actual parameter E_2

2 $E ::= \lambda N.E_1$ ← lambda abstraction (variable binding/function)

3 $E ::= N$ ← reference to formal parameter

The syntactic class E denotes the class of expressions in the language of Lambda Calculus. N is an infinite class of names, each distinguishable from the others. Names in production 3 are called *references*, while names used in production 2 are called definitions or *bindings*, and the used name, N, is said to be *bound* there, and is said to be *defined* throughout the body expression E_1. An additional constraint exists, namely that *outer* expressions (expressions not descended from another expression) be closed. This means that references must be occur only where they are defined. To remove ambiguity in association between references and definitions, I also require that no binding of a variable occur where the same variable name is already defined. [I avoid the usual discussion of shadowing]. Because the theory is extensional, the renaming of a variable in a binding and all references to the same name within its body expression, to another name which was not defined there, is considered an insignificant transformation, and is called α-conversion. The outer expression has the same meaning before and after the transformation.

A.1.1 β - conversion

β-conversion is the core operation of Lambda Calculus. It is the only significant transformation defined for expressions. It is strictly intensional, and changes only the form, and not the meaning, of a λ-expression.

There are two forms of β-conversion: (1) β-reduction, and (2) expansion, which reverses a β-reduction. For convenience, I define β-reduction only on expressions where all bindings are for unique names (a state easily achieved by α-conversions).

β-reduction of a function (written according to production 2), applied (according to production 1) to a parameter, substitutes, for the application, the body of the function, with all defined occurrences of the variable replaced with the actual parameter. Via β-reduction, the expression:

$(\lambda N_1.E_1\ E_2)$

reduces to E_1' where E_1' is E_1 with all occurrences of N_1 replaced with E_2.

Any outer expression may have the result of a β-reduction, where applicable, substituted for any application within the outer expression, without changing the meaning of the outer expression. This is fundamental to most lambda calculi, although there are a few that forbid reduction/expansion in certain sub-expressions. The result is that one may manipulate (Pure Lambda Calculus) expressions by freely performing β-reductions and expansions on applicable parts in order to transform the expression without changing its meaning. Thus, one may change: $((\lambda x.(x\ x)\ \lambda y.y)\ \lambda a.(a\ a))$ to: $((\lambda y.y\ \lambda y.y)(\lambda b.b\ \lambda a.(a\ a)))$ at will, since the sub-expression $(\lambda x.(x\ x)\ \lambda y.y)$ β-reduces to $(\lambda y.y\ \lambda y.y)$ and sub-expression $\lambda a.(a\ a)$ expands to $(\lambda b.b\ \lambda a.(a\ a))$ (because $(\lambda b.b\ \lambda a.(a\ a))$ reduces to $\lambda a.(a\ a)$).

A.1.2 Use of parentheses

I adopt the usual convention that application associates to the left, and I may drop some parentheses when ambiguity is not introduced. In particular, a subexpression such as "$(a\ b\ c\ d\ e)$" is an abbreviated form of "$((((a\ b)\ c)\ d)\ e)$". We may also insert unnecessary parenthesis (around complete syntactic units) for convenience or for clarity. So, an expression such as "x" may also be written as "(x)" or as "$((x))$".

A.1.3 Currying

Functions of multiple parameters may be simulated through *currying*, accepting the first parameter and then returning a function that takes the remaining parameters. Currying is applied recursively as necessary to simulate the number of parameters needed. In the following example, $\lambda x.\ \lambda y.$ (expression with x and y) simulates a function of 2 formal parameters, x, and y. The expression:

$$((\lambda x.\ \lambda y.\ \text{(expression with } x \text{ and } y))\ X\ Y)$$

simulates the application of this function to 2 actual parameters, X, and Y.

Starting with $((\lambda x.\ \lambda y.\ \text{(expression with } x \text{ and } y))\ X\ Y)$, we re-parenthesize the expression:
$((\lambda x.\ \lambda y.\ \text{(expression with } x \text{ and } y)\ X)\ Y)$, then apply $(\lambda x.\ \ldots)$ to X,
yielding the function $\lambda y.$ (expression with X and y) in:
$(\lambda y.\ \text{(expression with } X \text{ and } y)\ Y)$, and then one last application, producing:
(expression with X and Y), which is the expected result from applying the function of 2 parameters.

A.1.4 Equivalence of λ-terms in $Th(\mathcal{K}^*)$

λ-terms, related by α- or β-conversion, are by definition equivalent, and such equivalence is commutative and associative. There is only one "type" of λ-term. That is, every expression is a function, which takes a λ-expression as input, and returns a λ-expression as its result. In view of this, it is reasonable to ask if there is more than one "value", or if all λ-expressions are actually equivalent.

It should be noted that many applications of Lambda Calculus involve a mixture of Lambda Calculus and other mathematical notations, where more than one entity definitely exists. In these applications, the abstract syntax might be extended to include some additional production, such as:

4 $E ::= X$ ← a mention of some other kind of mathematical object

Suppose A and B denote different mathematical objects of the kind conforming to X. Then the expressions: $((\lambda x.\lambda y.xA)B)$ and $((\lambda x.\lambda y.yA)B)$ reduce, by β-conversion, to:
A and B, respectively.

Obviously, any notion of equivalence that equates $\lambda x.\lambda y.x$ with $\lambda x.\lambda y.y$ (within pure Lambda Calculus) would be difficult to extend with a production such as production 4. The expressions $((\lambda x.\lambda y.xA)B)$ and $((\lambda x.\lambda y.yA)B)$ would also be equivalent by substitution of equivalents, so that A and B must also be equivalent by β reductions. Hence, the desire to practically apply Lambda Calculus provides some indication of which expressions must be non-equivalent.

Here I provide a maximal extensional definition of equivalence, which preserves equivalence by α- or β-conversion, and which conforms to reasonable intuitions about terms which should be considered distinguishable. First, I define *head normal form*, then use that in the definition of equivalence.

A λ-expression is in head normal form iff it is derived through a sequence of production 2, followed by a sequence of production 1, when descending the E_1 branch, followed by production 3. This is exactly the set of expressions of the form $\lambda x_1.\ldots\lambda x_n.(x_i\ldots)$, for some $n \geq 1$, $i \in \{1\ldots n\}$. In this case, the variable reference x_i is said to be in the *head position*, and the expression $\lambda x_1.\ldots\lambda x_n(x_i\ldots)$, and the function it represents, is said to be *strict* in its ith parameter.

An expression e is said to *have a head normal form* iff e is related by a finite sequence of α- and β-conversions to some expression e' that is in head normal form.

Two expressions e_1 and e_2 are equivalent iff for every finite expression d, $(d\ e_1)$ has a head normal form iff $(d\ e_2)$ has a head normal form.

This definition is equivalent to the standard definition from [4]. Under this definition, all expressions having no head normal form are equivalent to each other. This unique value is named $\bot$, so we write, for example, $(\lambda x.(x\ x)\ \lambda x.(x\ x)) = \bot$ and $(\lambda y.(y\ y\ y)\ \lambda y.(y\ y\ y)) = \bot$.

A.1.5 Insignificance of α-conversion

For convenience, we consider α-conversion an insignificant operation, and do not discuss α-conversion steps. Thus, we may say $((\lambda x.(x\ x)\ \lambda x.x)(\lambda a.a\ \lambda b.(b\ b)))$ β-reduces to: $((\lambda y.y\ \lambda y.y)(\lambda a.a\ \lambda b.(b\ b)))$ which β-reduces to: $((\lambda y.y\ \lambda y.y)\ \lambda a.(a\ a))$, without mentioning the α-conversion steps involved.

A.1.6 Encodings

The following *Church encodings* may be meaningfully used to encode booleans, arithmetic, and data structures within Lambda Calculus. (A few of these are of my own devising; also note that Church's original paper[18] did not use Church encodings for numerals.)

1. $true = \lambda t.\lambda f.t$
2. $false = \lambda t.\lambda f.f$
3. $not = \lambda b.\lambda t.\lambda f.(b\ f\ t)$
4. $and = \lambda b_1.\lambda b_2.(b_1\ b_2\ b_1)$

5. $or = \lambda b_1.\lambda b_2.(b_1\ b_1\ b_2)$
6. $0 = \lambda f.\lambda x.x$
7. $1 = \lambda f.\lambda x.(f\ x)$
8. $2 = \lambda f.\lambda x.(f\ (f\ x))$
9. $3 = \lambda f.\lambda x.(f\ (f\ (f\ x)))$
10. $n = \lambda f.\lambda x.(\ldots n\ f's \ldots\ x\ \ldots)$
11. $\infty = \lambda f.\lambda x.((\lambda F.(F\ F))(\lambda F.(f\ (F\ F))))$
12. $times = \lambda a.\lambda b.\lambda f.(a(b\ f))$
13. $plus = \lambda a.\lambda b.\lambda f.\lambda x.((a\ f)((b\ f)x))$
14. $cons = \lambda head.\ \lambda tail.\ \lambda selector.\ (selector\ head\ tail)$
15. $car = \lambda cell.\ (cell\ true)$
16. $cdr = \lambda cell.\ (cell\ false)$

Under these encodings, the usual logical and arithmetic laws hold (for all $x, y, z \in \{0 \ldots \infty\}$, $(plus\ x\ y) = (plus\ y\ x)$, $(times(plus\ x\ y)z) = (plus(times\ x\ z)(times\ y\ z))$, etc.). Here I would like to explain the above definition of ∞. One would like to define ∞ as:

$$\infty = \lambda f.\lambda x.(\ldots \infty\ f's \ldots\ x\ \ldots),$$

following after the pattern in encoding 10, above. This definition would not technically be printable of course. I actually define ∞ with the following *recursive* definition:

$$\infty = \lambda f.\lambda x.(f((\infty f)x)).$$

Lambda Calculus, however has no recursive facility, so technically this is not a definition. Regardless, consider what happens if this definition $(\lambda f.\lambda x.(f((\infty f)x)))$ of ∞ is substituted for the "∞" in the expression $\lambda f.\lambda x.(f((\infty f)x))$. The resulting expression is $\lambda f.\lambda x.(f((\lambda f.\lambda x.(f((\infty f)x))f)x))$, and can then be reduced to the following: $\lambda f.\lambda x.(f(f((\infty f)x)))$. This recursive 'definition' clearly behaves as we would like, since we can continue to substitute $\lambda f.\lambda x.(f((\infty f)x))$ for ∞ in subsequent expressions, and reduce them, producing the following succession of 'definitions':

$$\infty$$

$$= \lambda f.\lambda x.(f((\infty f)x))$$

$$= \lambda f.\lambda x.(f(f((\infty f)x)))$$

$$= \lambda f.\lambda x.(f(f(f((\infty f)x))))$$

$$\ldots$$

$$= \lambda f.\lambda x.(\ldots \infty\ f's \ldots\ x\ \ldots).$$

The good news is that non-recursive definitions in Lambda Calculus have the same power as recursive definitions. For every recursive definition, such as $\infty = \lambda f.\lambda x.(f((\infty f)x))$, there exists an expression in Lambda Calculus having the same effect. Thus, the earlier definition 11 also has the desired behavior, as we see here:

$$\infty$$

$$= \lambda f.\lambda x.\underline{((\lambda F.(F\ F))(\lambda F.(f\ (F\ F))))}$$

$$= \lambda f.\lambda x.\underline{((\lambda F.(f\ (F\ F)))(\lambda F.(f\ (F\ F))))}$$

$$= \lambda f.\lambda x.((f\ \underline{((\lambda F.(f\ (F\ F)))(\lambda F.(f\ (F\ F))))}))$$

$$= \lambda f.\lambda x.((f\ ((f\ \underline{((\lambda F.(f\ (F\ F)))(\lambda F.(f\ (F\ F))))})))$$

$$= \lambda f.\lambda x.((f\ ((f\ ((f\ \underline{((\lambda F.(f\ (F\ F)))(\lambda F.(f\ (F\ F))))})))))$$

$$\ldots.$$

I have underlined the expression that is reduced in each step.

In general, a desired definition of the following form:

$$f = (Gf),$$

for any non-recursively defined expression G, may be rewritten, without the use of recursion, as follows:

$$f = ((\lambda F.(G(F\ F)))(\lambda F.(G(F\ F)))).$$

By reducing $((\lambda F.(G(F\ F)))(\lambda F.(G(F\ F))))$ in this definition, we have:

$$f = ((G(\underline{(\lambda F.(G(F\ F)))(\lambda F.(G(F\ F)))}))).$$

Recalling that the underlined expression is the body of our definition of f, we see that:

$$f = ((G(f)))$$

holds, as desired. Henceforth, I may use recursive definitions, with the understanding that they are abbreviated forms of proper non-recursive definitions.

A.1.7 Some theorems pertaining to the use of Lambda Calculus for Logic

These theorems are well known:

0 The 'i-rule': In any extensional theory of lambda calculus:

For any expression F:

$\lambda x.(F\ x) = F$.

1 There is no algorithm A such that:

For any finite encodings $\langle X\rangle$, $\langle Y\rangle$ of closed terms X, Y,

$((A\langle X\rangle)\langle Y\rangle) = (\lambda t.\lambda f.t)$ iff $X = Y$, $((A\langle X\rangle)\langle Y\rangle) = (\lambda t.\lambda f.f)$ otherwise

$\leftarrow$ Equivalence of λ-terms is generally undecidable, not encodable as a λ-term yielding a boolean.

2 For any closed terms X_1, Y_1, X_2 and Y_2, $(\lambda s.(sX_1)Y_1) = (\lambda s.(sX_2)Y_2) \leftrightarrow X_1 = X_2 \wedge Y_1 = Y_2$

$\leftarrow$ this may be used to encode a conjunction of equations as an equation.

For most of the following theorems, which are stated here without proof, I have unpublished proofs. In some cases, the theorems may have been previously published by others, however I have been unable to find any such publications.

3 For any closed term X, $(\lambda y.(X\ y\ y)) = (\lambda y.y) \leftrightarrow X \in \{(\lambda t.\lambda f.t), (\lambda t.\lambda f.f)\}$

$\leftarrow$ this may be used to encode the assertion that X has boolean type as an equation.

4 For any closed term X, $(plus\ X\ 1) = (plus\ 1\ X) \leftrightarrow X \in \{0, 1, 2, \ldots \infty\}$

$\leftarrow$ this may be used to encode the assertion that X has natural (including ∞) type as an equation.

5 There are no algorithms A_L, and A_R such that: For any finite encodings $\langle X\rangle$, $\langle Y\rangle$ of closed terms X and Y, $(A_L\langle X\rangle)\langle Y\rangle = (A_R\langle X\rangle)\langle Y\rangle \leftrightarrow X \neq Y$ $\leftarrow$ negation of equations may not be encoded by equations.

6 There are no algorithms A_L, and A_R such that:

For any finite encodings $\langle X_1\rangle$, $\langle Y_1\rangle$, $\langle X_2\rangle$ and $\langle Y_2\rangle$ of closed terms X_1, Y_1, X_2 and Y_2,

$((((A_L\langle X_1\rangle)\langle Y_1\rangle)\langle X_2\rangle)\langle Y_2\rangle) = ((((A_R\langle X_1\rangle)\langle Y_1\rangle)\langle X_2\rangle)\langle Y_2\rangle) \leftrightarrow X_1 = X_2 \vee Y_1 = Y_2$

$\leftarrow$ disjunction of equations may not be encoded by equations.

A.1.8 Continuation Passing Style

Continuations were originally used in studies of denotational semantics to model (in Lambda Calculus) the control state of sequential programs. It was eventually found that continuations were useful in compilation of functional programs to execute on sequential architectures. Continuation Passing Style (CPS) resembles tail recursion, in that it avoids stacking of return contexts, so that compiled functions do not return. Since CPS functions do not return, those computations that would be done after a non-CPS version of the function returns must be executed some other way. In CPS, those computations are packed into a continuation which is passed to the CPS function. After performing its computation, and computing its result, the CPS function applies the input continuation to the function result, thereby sending its result to the remainder of the computation, without returning.

Since there is no returning of functions (or anything else), currying is not used to simulate multiple parameters. Instead, multiple parameters are allowed directly. Thus we may write: $\lambda\langle x, y\rangle.(\ldots body \ldots)$ to denote a 2-parameter function in CPS notation.

For example, a function that returns the sum of 2 parameters: $\lambda a.\lambda b.(plus\ a\ b)$ would be transformed into CPS as a function with an additional parameter C, the continuation, which will perform the remainder of the computation. Thus, $\lambda a.\lambda b.(plus\ a\ b)$ might be translated into CPS as something like: $\lambda\langle a, b, C\rangle.(C\ (plus\ a\ b))$, so the result $(plus\ a\ b)$ is passed on to the continuation C. That example is not precise, however, and the actual CPS translation is more complex, because the intermediate result $(plus\ a)$ must also be passed to some continuation, as the currying of parameters a and b must be properly simulated. Also, the CPS translation of *plus* will accept a continuation parameter.

Although many uses have been found for CPS, the important observation, for the purposes of this proposal, is that any function may be converted into a form where the return mechanism is not involved in the computation, so that its return value type becomes irrelevant. This may seem somewhat counter-intuitive,

until one remembers that CPS transformation occurs as part of compilation, and that the compiled program is not strictly functional, and communicates its result to the outside world by a side effect, such as printing.

The CPS translation of a λ-expression is defined as follows: There are multiple ways to convert an expression to CPS. Some other theories require more complex conversion rules, often due to strictness requirements which rule out some reduction orders, but the following rules suffice for $Th(K^*)$.

1 $CPS[\![x]\!] = \lambda\langle c, z'\rangle.(x'\langle c, z'\rangle) = x'$

2 $CPS[\![(x\ y)]\!] = \lambda\langle c, z'\rangle.(CPS[\![x]\!]\ \langle\lambda q.q\langle c, z'\rangle, CPS[\![y]\!]\rangle)$

3 $CPS[\![\lambda z.y]\!] = \lambda\langle c, z'\rangle.(c\ CPS[\![y]\!])$

Here, $CPS[\![expr]\!]$ denotes the CPS conversion of *expr*. A primed variable denotes a variable bound in the CPS-converted expression, usually corresponding to an unprimed variable of the same name in the original expression.

Because every lambda expression takes one argument, every lambda expression is converted into a CPS expression taking a continuation and an argument. A practical exception to this must be an outer expression that produces the result of interest. We would like an outer expression to take only a continuation, to which the result is passed. To achieve this we will convert outer expressions according to the following rules:

4 $CPS[\![x]\!] = \leftarrow$ An outer expression cannot be an unbound variable

5 $CPS[\![(x\ y)]\!] = \lambda C.(CPS[\![x]\!]\ \langle C, CPS[\![y]\!]\rangle)$

6 $CPS[\![\lambda z.y]\!] = \lambda C.(C\ \lambda\langle c, z'\rangle.(c\ CPS[\![y]\!]))$

The following example is included to provide a minimal illustration of the use of CPS. I use primes also for CPS versions of encoded numerals and arithmetic operators, in lieu of displaying their entire converted form. Imagine an interactive interpreter for expressions that prints out the function result. The user types: $\lambda x.((plus\ x)1)7$, which applies the function $\lambda x.((plus\ x)1)$, which adds 1 to its input, to the encoded numeral 7. The interpreter converts the expression as follows:

$CPS[\![(\lambda x.((plus\ x)1)\ 7)]\!] = \lambda C.(CPS[\![\lambda x.((plus\ x)1)]\!]\ \langle C, CPS[\![7]\!]\rangle)$ $\leftarrow$ using rule 5.

$CPS[\![7]\!] = 7'$ $\leftarrow$ Written this way to avoid verbosity.

$CPS[\![\lambda x.((plus\ x)1)]\!] = \lambda\langle c_1, x'\rangle.(c_1\ CPS[\![((plus\ x)1)]\!])$ $\leftarrow$ using rule 3.

$CPS[\![((plus\ x)\ 1)]\!] = \lambda\langle c_2, z_2'\rangle.(CPS[\![(plus\ x)]\!]\ \langle\lambda q_2.q_2\langle c_2, z_2'\rangle, CPS[\![1]\!]\rangle)$ $\leftarrow$ using rule 2.

$CPS[\![1]\!] = 1'$ $\leftarrow$ Written this way to avoid verbosity.

$CPS[\![(plus\ x)]\!] = \lambda\langle c_3, z_3'\rangle.(CPS[\![plus]\!]\ \langle\lambda q_3.q_3\langle c_3, z_3'\rangle, CPS[\![x]\!]\rangle)$ $\leftarrow$ using rule 2.

$CPS[\![plus]\!] = plus'$ $\leftarrow$ Written this way to avoid verbosity.

$CPS[\![(\lambda x.((plus\ x)1)\ 7)]\!] =$

$\lambda C.(\lambda\langle c_1, x'\rangle.(c_1\ \lambda\langle c_2, z_2'\rangle.(\lambda\langle c_3, z_3'\rangle.(plus'\ \langle\lambda q_3.q_3\langle c_3, z_3'\rangle, x'\rangle)\ \langle\lambda q_2.q_2\langle c_2, z_2'\rangle, 1'\rangle))\ \langle C, 7'\rangle)$ $\leftarrow$ Putting all the substitutions together.

The interpreter uses a 'print' function as the continuation passed to this CPS expression, so the result will be printed:

$(\lambda C.(\lambda\langle c_1, x'\rangle.(c_1\ \lambda\langle c_2, z_2'\rangle.(\lambda\langle c_3, z_3'\rangle.(plus'\ \langle\lambda q_3.q_3\langle c_3, z_3'\rangle, x'\rangle)\ \langle\lambda q_2.q_2\langle c_2, z_2'\rangle, 1'\rangle))\ \langle C, 7'\rangle)\ print)$

We then imagine that computation can proceed by reductions and evaluations shown in red:

$(\lambda C.(\lambda\langle c_1, x'\rangle.(c_1\ \lambda\langle c_2, z_2'\rangle.(\lambda\langle c_3, z_3'\rangle.(plus'\ \langle\lambda q_3.q_3\langle c_3, z_3'\rangle, x'\rangle)\ \langle\lambda q_2.q_2\langle c_2, z_2'\rangle, 1'\rangle))\ \langle C, 7'\rangle)\ print)$

$(\lambda\langle c_1, x'\rangle.(c_1\ \lambda\langle c_2, z_2'\rangle.(\lambda\langle c_3, z_3'\rangle.(plus'\ \langle\lambda q_3.q_3\langle c_3, z_3'\rangle, x'\rangle)\ \langle\lambda q_2.q_2\langle c_2, z_2'\rangle, 1'\rangle))\ \langle print, 7'\rangle)$

$(\lambda\langle c_1, x'\rangle.(c_1\ \lambda\langle c_2, z_2'\rangle.(\lambda\langle c_3, z_3'\rangle.(plus'\ \langle\lambda q_3.q_3\langle c_3, z_3'\rangle, x'\rangle)\ \langle\lambda q_2.q_2\langle c_2, z_2'\rangle, 1'\rangle))\ \langle print, 7'\rangle)$
$(print\ \lambda\langle c_2, z_2'\rangle.(\lambda\langle c_3, z_3'\rangle.(plus'\ \langle\lambda q_3.q_3\langle c_3, z_3'\rangle, 7'\rangle)\ \langle\lambda q_2.q_2\langle c_2, z_2'\rangle, 1'\rangle))$
$(print\ \lambda\langle c_2, z_2'\rangle.(\lambda\langle c_3, z_3'\rangle.(plus'\ \langle\lambda q_3.q_3\langle c_3, z_3'\rangle, 7'\rangle)\ \langle\lambda q_2.q_2\langle c_2, z_2'\rangle, 1'\rangle))$
$(print\ \lambda\langle c_2, z_2'\rangle.(plus'\ \langle\lambda q_3.q_3\langle\lambda q_2.q_2\langle c_2, z_2'\rangle, 1'\rangle, 7'\rangle))$
$(print\ \lambda\langle c_2, z_2'\rangle.(plus'\ \langle\lambda q_3.q_3\langle\lambda q_2.q_2\langle c_2, z_2'\rangle, 1'\rangle, 7'\rangle))$ ← Let $add7$ = the result of applying $plus$ to 7.
$(print\ \lambda\langle c_2, z_2'\rangle.(\lambda q_3.q_3\langle\lambda q_2.q_2\langle c_2, z_2'\rangle, 1'\rangle\ add7'))$ ← The result $add7'$ is passed to the continuation.
$(print\ \lambda\langle c_2, z_2'\rangle.(\lambda q_3.q_3\langle\lambda q_2.q_2\langle c_2, z_2'\rangle, 1'\rangle\ add7'))$
$(print\ \lambda\langle c_2, z_2'\rangle.(add7'\langle\lambda q_2.q_2\langle c_2, z_2'\rangle, 1'\rangle))$
$(print\ \lambda\langle c_2, z_2'\rangle.(add7'\langle\lambda q_2.q_2\langle c_2, z_2'\rangle, 1'\rangle))$ ← Let 8 = the result of applying $add7$ to 1.
$(print\ \lambda\langle c_2, z_2'\rangle.(\lambda q_2.q_2\langle c_2, z_2'\rangle\ 8'))$ ← The result $8'$ is passed to the continuation.
$(print\ \lambda\langle c_2, z_2'\rangle.(\lambda q_2.q_2\langle c_2, z_2'\rangle\ 8'))$
$(print\ \lambda\langle c_2, z_2'\rangle.(8'\langle c_2, z_2'\rangle))$
$(print\ 8')$ ← Via the i-rule.

Although interpreters usually do not implement the i-rule, I take this liberty to simplify the presentation.

A.2 Lambda Predicate Logic

Here I describe the proposed core language for expression of logical statements. Lambda Predicate Logic has both statements and predicates, instead of encoding logical values as functions, or using UR-elements[6] to introduce logical entities such as booleans into the pure λ-calculus.

I instead have a distinct syntactic category, *Sentence*, for logical statements, and use λ notation only as a way to write anonymous predicates, rather than functions. The lack of functions removes the possibility of currying, so predicates instead have multiple parameters.

Here is an 'abstract syntax' of the language of Lambda Predicate Logic:

1. *Sentence* ::= Conjunction *Sentence** ← Conjunction of 0 or more *Sentence*s
2. *Sentence* ::= Negation *Sentence* ← Negation of a *Sentence*
3. *Sentence* ::= Disjunction *Sentence** ← Disjunction of 0 or more *Sentence*s (needed for intuitionistic developments)
4. *Sentence* ::= Application *Predicate* (*Name* = *Predicate*)* ← 'name list' style parameter passing
5. *Predicate* ::= Lambda *Sentence* ← The body of a *Predicate* is a *Sentence*
6. *Predicate* ::= ParameterReference *Name PredicateIndex* ← Parameter reference
7. *Predicate* ::= Undefined ← Meaningless *Predicate* filler

It is important here to note that I am describing just the core language, and not the core logical system. *Sentence*s may be combined via a minimal set of logical operators (Conjunction and Negation), to which I add Disjunction, for use in intuitionistic logical developments (productions 1,2,3). I add no additional operators for supporting intuitionistic logic, although Implication might appear to be highly desirable.

The atomic *Sentence*s are Applications of *Predicate*s to arguments, all of which are *Predicate*s (production 4). The argument list associates the collection of *Name*s with *Predicate* 'values'. Every reference to a parameter is therefore also a *Predicate* (production 6).

[6] I use 'UR-elements' to mean primitive objects defined externally to the system under discussion

Anonymous *Predicate*s may be defined using Lambda notation, where the body of such a Lambda expression is a *Sentence* (production 5). Similarly to the Lambda Calculus, the formal parameters of a *Predicate* may be referenced within its body *Sentence*. Since every *Predicate* has multiple named formal parameters, and anonymous *Predicate* definitions may be nested to any depth, a $ParameterReference$ (production 6) must have two parts: (1) a *Name*, indicating which formal parameter of the anonymous *Predicate* definition is referenced, and (2) a *PredicateIndex*, indicating which nested anonymous *Predicate* definition has the formal parameter being referenced.

The set *Name* is required to have at least two elements, while the set of *PredicateIndex*es must have a countably infinite number of elements. Every Application must include parameter assignments for all names, however, in the present development, I assume the usual countably infinite set of names, as is usually convenient for Lambda Calculus. Since, for any given predicate definition, usually only a small finite set of names are of interest, the remaining names must be assigned a meaningless value. The Undefined *Predicate* (production 7) is provided for this purpose, and is used as a syntactic filler, not considered to supply an actual predicate. Conveniently, we may write an incomplete list of parameters, as an abbreviation for a complete list, where the parameters not explicitly given are assumed to be assigned the 'Undefined' value. Although this complication of 'name list' style parameter passing and reference appears to be awkward, the developments herein will be made readable by the use of a presentation syntax (Section A.2.2) having a conventional appearance.

Finally, I require a specific pre-defined *Predicate* for Equivalence of *Predicate*s:

8. *Predicate* ::= Equals ← A predefined *Predicate*

Although predefined *Predicate*s are more properly discussed in the context of logical systems, there is only this one, so, for convenience, I describe it here as part of the language.

A.2.1 Concrete Syntax

Here is a concrete syntax I will use herein, only when necessary, for the language of Lambda Predicate Logic:

1. *Sentence* ::= '{' *Sentence** '}' ← Conjunction of 0 or more *Sentence*s
2. *Sentence* ::= ' ¬ ' *Sentence* ← Negation of a *Sentence*
3. *Sentence* ::= '⋁{' *Sentence** '}' ← Disjunction of 0 or more *Sentence*s
4. *Sentence* ::= *Predicate* '{' ('|'*Name*' => '*Predicate*')* '}' ← 'name list' style parameter passing
5. *Predicate* ::= 'λ'$_{PredicateIndex}$ *Sentence* ← The body of a *Predicate* is a *Sentence*
6. *Predicate* ::= $Name_{PredicateIndex}$ ← Parameter reference
7. *Predicate* ::= 'Undefined' ← Meaningless *Predicate* filler
8. *Predicate* ::= 'Equals' ← A predefined equivalence *Predicate*

These productions correspond to the abstract syntax described before. Thus, we may write:

$Equals\{|lhs => \lambda_1 x_1\{|arg => y_1\ \}\ |rhs => \lambda_1 x_1\{|arg => y_1\ \}\ \}$, meaning that the *Equals* predicate holds when the *lhs* and *rhs* arguments are both passed the predicate value: $\lambda_1 x_1\{|arg => y_1\ \}$. The *Predicate* $\lambda_1 x_1\{|arg => y_1\ \}$ is parsed as follows. It is derived from production 5, and has the *PredicateIndex* 1. Its body *Sentence*, "$x_1\{|arg => y_1\ \}$", derived from production 4, applies the

predicate x_1 to the argument list $\{|arg => y_1\ \}$, which assigns the *Predicate* value y_1 to the parameter *arg*. The *Predicate* value y_1, finally, is a formal Parameter reference, for the 'y' parameter of the *Predicate* $\lambda_1 x_1\{|arg => y_1\ \}$. The association between the parameter reference y_1, and the *Predicate* $\lambda_1 x_1\{|arg => y_1\ \}$ is inferred by the reference being contained in the body of the *Predicate*, and the common *PredicateIndex* subscript "$_1$" on both the parameter reference y_1 and the lambda λ_1 of the *Predicate*.

Note that $x_1\{|arg => y_1\ \}$ is an abbreviation for

$$x_1\{|a => Undefined\ |b => Undefined\ |c => Undefined\ \ldots|arf => Undefined$$

$$|arg => y_1$$

$$|arh => Undefined\ \ldots|anotherbigname => Undefined\ \ldots\},$$

that is, every possible parameter name is given, but the parameters that are not of interest are assigned the 'value' *Undefined*.

The *Sentence* $Equals\{|lhs => \lambda_1 x_1\{|arg => y_1\ \}\ |rhs => \lambda_1 x_1\{|arg => y_1\ \}\ \}$ is a *Predicate* application (production 4) of the *Predicate Equals* (production 8), with the arguments, *lhs* and *rhs* both set to the value $\lambda_1 x_1\{|arg => y_1\ \}$.

The following example illustrates the purpose of the *PredicateIndex* subscript on parameter references and on λs. As applications include parameter assignments for all *Name*s, so all *Name*s are defined within each *Predicate* body. This creates ambiguity without the use of *PredicateIndex*s. In the *Sentence* $\lambda_1\lambda_2 x_1\{\ldots\}\{|x => A\ \}\{|x => B\ \}$, the *Predicate* $\lambda_1\lambda_2 x_1\{\ldots\}\{|x => A\ \}$ is applied to the argument list $\{|x => B\ \}$, and the *Predicate*s body is the *Sentence* $\lambda_2 x_1\{\ldots\}\{|x => A\ \}$ (A and B are some arbitrary *Predicate*-valued expressions with no free variables). In turn, the *Sentence* $\lambda_2 x_1\{\ldots\}\{|x => A\ \}$ is an application of the *Predicate* $\lambda_2 x_1\{\ldots\}$ to the argument list $\{|x => A\ \}$. The body ($x_1\{\ldots\}$) of this *Predicate*, thus is nested within two different anonymous *Predicate* definitions, so that the referent of the reference x would be ambiguous, since the *Name* x is defined within both *Predicate* bodies. The *PredicateIndex* '$_1$' attached to the reference x_1 indicates it refers to the x parameter of the *Predicate* $\lambda_1\lambda_2 x_1\{\ldots\}\{|x => A\ \}$, which has the same *PredicateIndex* attached to its λ_1.

A.2.2 Presentation Syntax

As the above concrete syntax is cumbersome and unfamiliar, and the abstract syntax is not viewable, I define a presentation syntax, which I consider to be an abbreviated shorthand for the equivalent concrete syntax. I will use the following informally defined syntax for most of my discussion here, using the concrete syntax only when necessary for clarity. The main simplification will avoid the use of the unfamiliar *PredicateIndex*es in most cases, and the use of name-list notation in actual parameter lists. Additionally, equation syntax and logical operator syntax will be more familiar.

As all predicate definitions currently of interest use only a few parameters, I will list the names (in a chosen order) of those parameters after the λ of the definition, as a tuple enclosed by angle brackets ($\langle\rangle$). Thus, I may write $\lambda\langle x\rangle\lambda\langle\rangle x\{\ldots\}\{|x => A\ \}\{|x => B\ \}$ instead of $\lambda_1\lambda_2 x_1\{\ldots\}\{|x => A\ \}\{|x => B\ \}$. Notice that the *PredicateIndex* was dropped from the reference x, since it is no longer needed to avoid ambiguity. When the parameter list of a predicate is known at the point of application, I may write the actual parameters as a tuple enclosed in angle brackets. In this case, the actual parameters must be given in the same order as the parameter list, so the parameter names need not be given redundantly. Thus, I may write $\lambda\langle x\rangle\lambda\langle\rangle x\{\ldots\}\{|x => A\ \}\langle B\rangle$ in place of the above, since the parameter list $\langle x\rangle$ is given for the *Predicate* $\lambda\langle x\rangle\lambda\langle\rangle x\{\ldots\}\{|x => A\ \}$. Note that I still write $\{|x => A\ \}$for the actual parameters of

$\lambda\langle\rangle x\{\ldots\}$, since it specifies a value for a parameter x that is not actually used in the predicate, and thus cannot be syntactically simplified. I may also decorate a name in a formal parameter list with a superscript comma-separated list of the formal parameters used by the predicate value of that name, increasing the opportunities for simplification. Thus, instead of $\lambda\langle x\rangle\lambda\langle\rangle x\{|p => A\ |q => B\ \}\{|x => A\ \}\langle B\rangle$, I may write: $\lambda\langle x^{\langle p,\ q\rangle}\rangle\lambda\langle\rangle x\langle A,\ B\rangle\{|x => A\ \}\langle B\rangle$, since the parameter list of the value x is given in the second version. Also, I actually may optionally include unused parameter names in a formal parameter list, allowing the above to be simplified as $\lambda_1\langle x^{\langle p,\ q\rangle}\rangle\lambda\langle x\rangle x_1\langle A,\ B\rangle\langle A\rangle\langle B\rangle$. Note that when a parameter reference becomes ambiguous due to being listed in parameter lists for multiple nested *Predicate* definitions, it becomes necessary to revert to use of *PredicateIndex*es to resolve the ambiguity. Finally, I use parentheses to indicate grouping around complete syntactical units, to improve clarity, so the above simplified expression may be written: $\lambda_1\langle x^{\langle p,\ q\rangle}\rangle(\lambda\langle x\rangle(x_1\langle A,\ B\rangle)\langle A\rangle)\langle B\rangle$.

I additionally allow use of infix logical operator notation in place of the prefix notation presented in the concrete syntax, and the use of the infix '=' sign in place of applications of the *Equals Predicate*, using parenthesis when necessary to clarify association.

Thus, I may conveniently write: $(\lambda\langle x^{\langle arg\rangle},\ y\rangle x\langle y\rangle) = (\lambda\langle x^{\langle arg\rangle},\ y\rangle x\langle y\rangle)$, as an abbreviation for the concrete syntax:

$$Equals\{|lhs => \lambda_1 x_1\{|arg => y_1\ \}\ |rhs => \lambda_1 x_1\{|arg => y_1\ \}\ \}$$

I may also write $(\neg A)\wedge(B\vee C)$ instead of $\{\neg A\ \bigvee\{B\ C\}\}$ (where A, B, and C are arbitrary sentences).

Thus, the presentation form of Lambda Predicate Logic differs from standard notation primarily in the use of angle brackets ($\langle\ \rangle$) for enclosing argument lists.

A.2.3 Semantics of Lambda Predicate Logic

Lambda Predicate Logic is intended to support meta-logic and equational reasoning[§5.10], in addition to formal specification, while exploiting the flexibility of lambda-calculus-like notation to simplify the language. Here I describe the semantics of Lambda Predicate Logic, using the less formal presentation syntax outlined above.

Alpha-conversion of *PredicateIndex*es: As with names in Lambda Calculus, alpha-conversion (renaming of all occurrences of a specific *PredicateIndex* that occur within its binding, to an unused *PredicateIndex*) preserves meanings and is considered an insignificant adjustment, so it will not be mentioned.

Alpha conversion of formal parameter *Name*s: Since a formal parameter *Name*, in some sense, denotes an interface to access a specific element of a tuple of parameters, there is no alpha conversion of formal parameter *Name*s.

Beta-conversion: As with Lambda Calculus, beta-conversion is the fundamental meaning-preserving transformation. I define it only informally here, via examples in the presentation syntax. As in Lambda Calculus, reduction and its inverse preserve meaning. In the application *Sentence*:

$$\lambda\langle i, f^{\langle n\rangle}, g^{\langle k\rangle}\rangle(f\langle i\rangle\wedge g\langle i\rangle)\langle 5, prime, small\rangle,$$

$\lambda\langle i, f^{\langle n\rangle}, g^{\langle k\rangle}\rangle(f\langle i\rangle\wedge g\langle i\rangle)$ is the predicate, and $\langle 5, prime, small\rangle$ is the argument list. The body of the predicate is $(f\langle i\rangle\wedge g\langle i\rangle)$, while $\langle i, f^{\langle n\rangle}, g^{\langle k\rangle}\rangle$ indicates which arguments are expected in this context. Thus the formal parameter i is given the actual parameter 5, f is given the actual parameter *prime*, and g is given the actual parameter *small*. Also, f, when used as a predicate, expects the formal parameter n, while g expects the formal parameter k. So, after reduction, the expression is:

$$(prime\langle 5\rangle\wedge small\langle 5\rangle),$$

Tab. 1: Encodings for Lambda Predicate Logic

traditional form	description	Lambda Predicate Logic
true	true, a conjunction of 0 conjuncts	$\{\}$
$A \Rightarrow B$ *[or* $(\neg A) \vee B$*]*	implication, as used for derivation	$(\lambda\langle\rangle A) = (\lambda\langle\rangle(A \wedge B))$
$\forall F : P(F)$	universal quantification over predicates	$P = \lambda\langle\rangle\{\}$
$\exists F : P(F)$ *[or* $\neg\forall F : \neg P(F)$*]*	existential quantification over predicates	$\neg((\lambda\langle F\rangle\neg P\langle F\rangle) = \lambda\langle\rangle\{\})$
$\forall s \in S : P(s)$	quantification over a set S	$S\langle P\rangle$ *[or* $S\langle\lambda\langle s\rangle P\langle s\rangle\rangle$*]*
$s \in S$	membership in set S	$S = \lambda\langle P\rangle(S\langle P\rangle \wedge P\langle s\rangle)$
[or $S = S \cup \{s\}$*]*	”	*[or* $S\langle P\rangle = S\langle P\rangle \wedge P\langle s\rangle$*]*

where *prime* should have a definition beginning with $\lambda\langle n\rangle$, indicating it takes the formal parameter n, and *small* has a definition beginning with $\lambda\langle k\rangle$, and 5 is assumed to be a suitable encoding of the natural number 5.

Definition of equivalence: The definition of equivalence for Lambda Predicate Logic depends not only on the language, but also on the logical system chosen, just as the definition of equivalence in lambda calculus depends on the chosen model. For reasons explained in Section 3, I prefer to choose a class of logical systems, rather than a specific logical system. Hence I only sketch a partial definition of equivalence via the following constraint list. The primary constraints I impose on logical systems are the following:

1. Beta-conversion equivalence: Predicates related by beta-conversion are equivalent.
2. Alpha-conversion equivalence: Predicates related by alpha-conversion of *PredicateIndex*es are equivalent.
3. Equational reasoning[§5.10]: A proven equation provides equal predicates, and substitution of equal predicates may be performed without changing meaning.
4. Non-trivialness: The predicate that always returns true ($\lambda\langle\rangle\{\}$) is not equivalent to the predicate that always returns false ($\lambda\langle\rangle\neg\{\}$).

A.2.4 Encodings for Lambda Predicate Logic

CPS conversion translates any Lambda Calculus expression into a *Predicate* in Lambda Predicate Logic, thus providing Lambda Predicate Logic with encodings for arithmetic, data structures, and all other objects expressible in Lambda Calculus. As Lambda Predicate Logic is intended to be more convenient for expressing specifications, encodings also exist for predicate-calculus-like expressions, and expressions involving set theories. A few of these are listed[7] in Table 1. Note that derivations, through modus ponens, in most systems use an implication symbol, while the equational reasoning of Lambda Predicate Logic requires useful implications to be encoded as equations.

Note that my encoding of sets is the opposite of that usually chosen in combinatory logics. I encode sets as a predicate which asserts a (parametric) predicate over all the members of the set, whereas combinatory logics usually encode sets as characteristic functions, which must be defined over a larger domain.

[7] S is encoded as a quantifier as if: $S\langle p\rangle = (p\langle s_0\rangle \wedge p\langle s_1\rangle \wedge p\langle s_2\rangle \wedge \ldots)$ where the s_0, s_1, s_2, …, are the members of S

A.2.5 Types for Lambda Predicate Logic

Although Lambda Predicate Logic is intrinsically typeless, I offer a simple system of types, which, like Lambda Predicate Logic itself, is somewhat independent of the logical/mathematical system chosen. A type of a *Sentence* V is also a *Sentence* T, and this typing may be written $V : T$. Given $V : T$, wherever the *Sentence* T holds, there V is either $true$ or $false$. Thus, $V : T$ is an abbreviated way of writing (using traditional notation):

$$T \Rightarrow (V \vee \neg V),$$

or, encoded in Lambda Predicate Logic:

$$\lambda\langle\rangle T = \lambda\langle\rangle(T \wedge (V \vee \neg V)),$$

Hence, this type system is not an extension of the language, but an added convenience. One *Sentence* may have many types. A type of a *Predicate* is, of course, another *Predicate*.

The meaning of a typing is especially significant when an intuitionistic system of logic is used. In this case, a typing indicates a circumstance where a sentence may be treated as a classical sentence, having either a $true$ or $false$ valuation. In these cases, the ability to reason is enhanced due to the additional information provided by the typing.

If a classical logic system is used, typing is still meaningful, since even some classical systems expressed within this language may admit undefined sentences. In this case, a type indicates when a sentence is defined, and is analogous to a condition guaranteeing termination in an executable language, such as Agda.

A.3 Old Ephemeral Implementation

The current implementation of Ephemeral is available for download at:

http://www.cs.ucr.edu/∼mummem/ephemeral.tar.gz .

(See http://www.cs.ucr.edu/∼mummem/Designdraft6.pdf (Ephemeral project compiler design document) for additional discussion of the current implementation)

The current implementation of Ephemeral is implemented as a translator to C++, also having a runtime component written in C++. Both compiler and compiled code run sequentially on Linux, with the runtime component providing allocation of places at space-time coordinates, storage management, and checking of message intervals.

The current implementation of Ephemeral has the following variations from the proposed Ephemeral described in Section 2.3.

- A specific built in four-dimensional coordinate system is used to reference places. The coordinate system is such that each axis has a time component and a space component, resulting in equal treatment of all axes. For any pair of coordinates $X = \langle x_0, x_1, x_2, x_3\rangle$, $Y = \langle y_0, y_1, y_2, y_3\rangle$, if $x_i \leq y_i$ for all $i \in \{0..3\}$, then Y is treated as being in a time-like interval after X, so a place at Y may receive a message sent from a place at X.

- No bound on the number of places co-located at a given coordinate is actually enforced, so the current implementation allows an unbounded quantity of computation to occur at a single coordinate.

- No distinction is made between connected pairs of places and other pairs, so that intermediate places are never needed to relay messages.

- All bit types are currently represented as 'long int' types in C++, regardless of their intended size.

- Action values are currently represented as code pointers.
- The 'strictly-time-like' relation required for intervals of messages containing results is not enforced. Instead, same 'time-like' relation required for other message intervals is enforced.
- User manipulation of space-time location pools is not currently supported.
- No notion of obligations of actions is supported. Hence, the restriction against copying or destroying port references is not enforced. Instead, before the computation of a place is started, the runtime checks that each port has received exactly one message.
- Transfer of a reference into an inappropriate coordinate is not checked. Instead, the interval between a sender and receiver is checked when a message is sent.
- Actions currently cannot trigger other actions in the same place.
- Only the first tuple item in a message may be the designated action of an active message.
- Most importantly, subtype relations are not currently supported, except what occurs naturally between action types.

A.3.1 Current Ephemeral Syntax

My implementation of the current version of Ephemeral has the following syntax:

- *Program*::=*declaration** ← program comprised of declarations
- *declaration*::=bit *name*[*expr*]; ← *name* is an arithmetic type of *expr* bits
- *declaration*::=code *name*{$name^*_{action}$}; ← code type *name* is a subset of actions
- *declaration*::=place *name*[: ...]{*port_declaration**}; ← *name* is aPlace type with declared ports
- *port_declaration*::=[− >]*name* : ($formal$[, $formal$]*); ← port *name* is a tuple of a $formal$s; '− >' indicates active message
- $formal$::=$name_{type}$ *name* ← typename, name
- *declaration*::= action $name(name_{place})${($allocation\_declaration$ | *send*)*}; ← executable source code can only allocate places and send messages, the members of which may be computed. The declared action can only be executed in a place of type $name_{place}$.
- $allocation\_declaration$::= $name_{placetype}$ $name_{place}$[($allocation\_specification$)]; ← Place allocation. Allocates $name_{place}$ as a new place of type $name_{placetype}$. The $allocation\_specification$ gives the coordinates of $name_{place}$ relative to the place hosting the current action.
- *send*::=$expr_{reference}$: (*expr*[, *expr*]*); ← send a message (the *expr*s) to a port ($expr_{reference}$) The *expr*s produce the values of tuple elements comprising the message. The number and types of the *expr*s must be the same (respectively) as the number and types of the formals declared in the *port_declaration* corresponding to the port referenced by $expr_{reference}$.
- *declaration*::=let *name* = *expr* ; ← convenience allowed in actions and some other places

- $expr ::= name_{place}.name_{port}$ ← reference a port
- $expr ::= name_{action}$ ← action name means code value
- $expr ::= name_{formal}$ ← field of an accessible message
- $expr ::= literal$ ← usual literal expressions
- $expr ::= operator_{unary}\ expr$ ← usual unary operator expressions
- $expr ::= expr\ operator_{binary}\ expr$ ← usual binary operator expressions
- $expr ::= expr\ ?\ expr\ :\ expr$ ← also includes trinary conditional

'let' declarations are allowed in bodies of actions as well as in the set of global declarations. The types of names so defined are inferred automatically. The ordering of all declarations is irrelevant, but globally defined type, action, and value names may not be defined in multiple declarations. There must be no cyclic dependencies among let declarations. 'let' declarations in the global set define values visible throughout the program, and may only refer to values visible throughout the entire program. 'let' declarations in the body of an action may refer to the tuple elements of any message in the place hosting the action, as well as to other values (such as references to the ports of created places) defined within the same action body. The defined values may be used only within the action body containing the definition. 'let' declarations are semantically equivalent to macros, however, in the current implementation, they may be used to force common calculation of common subexpressions. The current implementation calculates these values at compile time if the values have no dependence on non-literals.

References

[1] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., 2007.

[2] E. Allen, J. Hilburn, S. Kilpatrick, V. Luchangco, S. Ryu, D. Chase, and G. Steele. Type checking modular multiple dispatch with parametric polymorphism and multiple inheritance. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 973–992, New York, NY, USA, 2011. ACM.

[3] R.-J. J. Back, A. Akademi, and J. V. Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1998.

[4] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics, Revised*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1985.

[5] G. Barrett. Formal methods applied to a floating-point number system. *IEEE Trans. Softw. Eng.*, 15(5):611–621, May 1989.

[6] M. Berger. Specification and verification of meta-programs. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, PEPM '12, pages 3–4, New York, NY, USA, 2012. ACM.

[7] M. Berger and L. Tratt. Program logics for homogeneous meta-programming. In *Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR'10, pages 64–81, Berlin, Heidelberg, 2010. Springer-Verlag.

[8] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[9] P. Bone, Z. Somogyi, and P. Schachte. Controlling loops in parallel mercury code. In *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*, DAMP '12, pages 11–20, New York, NY, USA, 2012. ACM.

[10] A. Bove and P. Dybjer. Dependent types at work. In A. Bove, L. S. Barbosa, A. Pardo, and J. S. Pinto, editors, *Language Engineering and Rigorous Software Development*, pages 57–99. Springer-Verlag, Berlin, Heidelberg, 2009.

[11] S. Burckhardt, D. Leijen, C. Sadowski, J. Yi, and T. Ball. Two for the price of one: a model for parallel and incremental computation. *SIGPLAN Not.*, 46(10):427–444, Oct. 2011.

[12] E. Burrows and M. Haveraaen. Programmable data dependencies and placements. In *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*, DAMP '12, pages 31–40, New York, NY, USA, 2012. ACM.

[13] J. Carette and A. Stump. Towards typing for small-step direct reflection. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, PEPM '12, pages 93–96, New York, NY, USA, 2012. ACM.

[14] J. Caska and M. Schoeberl. Java dust: how small can embedded java be? In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, pages 125–129, New York, NY, USA, 2011. ACM.

[15] M. M. Chabbi, J. M. Mellor-Crummey, and K. D. Cooper. Efficiently exploring compiler optimization sequences with pairwise pruning. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 34–45, New York, NY, USA, 2011. ACM.

[16] T. W. Christopher. Communicating reactive objects: message-driven parallelism. *SIGPLAN Not.*, 37(4):27–28, Apr. 2002.

[17] T. W. Christopher. A simple parallel system. *SIGPLAN Not.*, 38(6):6–8, June 2003.

[18] A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[19] K. Claessen, M. Sheeran, and B. J. Svensson. Expressive array constructs in an embedded gpu kernel programming language. In *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*, DAMP '12, pages 21–30, New York, NY, USA, 2012. ACM.

[20] R. Cledat, K. Ravichandran, and S. Pande. Leveraging data-structure semantics for efficient algorithmic parallelism. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 28:1–28:10, New York, NY, USA, 2011. ACM.

[21] W. Clocksin and C. Mellish. *Programming in Prolog*. Springer-Verlag, 1981.

[22] A. Colmerauer. *My Happy Contribution to Constraint Programming (not in proceedings)*. Lecture Notes in Computer Science / Programming and Software Engineering. Springer, 2008.

[23] D. Crocker. Perfect developer: A tool for object-oriented formal specification and refinement. tools exhibition notes at formal methods europe. In *In Tools Exhibition Notes at Formal Methods Europe*, page 2003, 2003.

[24] O. de Moor, G. Gottlob, T. Furche, and A. J. Sellers, editors. *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, volume 6702 of *Lecture Notes in Computer Science*. Springer, 2011.

[25] D. Devriese and F. Piessens. On the bright side of type classes: instance arguments in agda. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 143–155, New York, NY, USA, 2011. ACM.

[26] A. Floch, T. Yuki, C. Guy, S. Derrien, B. Combemale, S. Rajopadhye, and R. B. France. Model-driven engineering and optimizing compilers: a bridge too far? In *Proceedings of the 14th international conference on Model driven engineering languages and systems*, MODELS'11, pages 608–622, Berlin, Heidelberg, 2011. Springer-Verlag.

[27] G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 163–175, New York, NY, USA, 2011. ACM.

[28] G. Goumas, S. A. McKee, M. Själander, T. R. Gross, S. Karlsson, C. W. Probst, and L. Zhang. Adapt or become extinct!: the case for a unified framework for deployment-time optimization (position paper). In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 46–51, New York, NY, USA, 2011. ACM.

[29] A. Grabowski, A. Kornilowicz, and A. Naumowicz. Mizar in a nutshell. *Journal of Formalized Reasoning*, 3(2):153–245, 2010.

[30] G. W. Hamilton and N. D. Jones. Distillation with labelled transition systems. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, PEPM '12, pages 15–24, New York, NY, USA, 2012. ACM.

[31] R. Harrop. Concerning formulas of the types $A \to B \wedge C$, $A \to (Ex)B(x)$ in intuitionistic formal systems. *J. Symb. Log.*, 25(1):27–32, mar 1960.

[32] C. T. Haynes and D. P. Friedman. Engines build process abstractions. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 18–24, New York, NY, USA, 1984. ACM.

[33] W. A. Howard. The formulae-as-types notion of construction. In H. Curry, J. Hindley, and J. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Boston, MA*, pages 479–490. Acad. Press, 1980.

[34] E. K. Jackson, T. Levendovszky, and D. Balasubramanian. Reasoning about metamodeling with formal specifications and automatic proofs. In *Proceedings of the 14th international conference on Model driven engineering languages and systems*, MODELS'11, pages 653–667, Berlin, Heidelberg, 2011. Springer-Verlag.

[35] G. Kimmell, A. Stump, H. D. Eades, III, P. Fu, T. Sheard, S. Weirich, C. Casinghino, V. Sjöberg, N. Collins, and K. Y. Ahn. Equational reasoning about programs with general recursion and call-by-value semantics. In *Proceedings of the sixth workshop on Programming languages meets program verification*, PLPV '12, pages 15–26, New York, NY, USA, 2012. ACM.

[36] R. Kirner, F. Penczek, and A. Shafarenko. Compilers must speak properties, not just code: Cal: constraint aggregation language for declarative component-coordination. In *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*, DAMP '12, pages 51–54, New York, NY, USA, 2012. ACM.

[37] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Software synthesis procedures. *Commun. ACM*, 55(2):103–111, Feb. 2012.

[38] J. Launchbury. Theorem-based circuit derivation in cryptol. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, GPCE '11, pages 185–186, New York, NY, USA, 2011. ACM.

[39] E. A. Lee. Heterogeneous actor modeling. In *Proceedings of the ninth ACM international conference on Embedded software*, EMSOFT '11, pages 3–12, New York, NY, USA, 2011. ACM.

[40] Y. Li and G. Dos Reis. An automatic parallelization framework for algebraic computation systems. In *Proceedings of the 36th international symposium on Symbolic and algebraic computation*, ISSAC '11, pages 233–240, New York, NY, USA, 2011. ACM.

[41] I. Limited. *Transputer Reference Manual*. Prentice Hall, London, 1988. ISBN 0-13-929001-X.

[42] C. Mead and L. Conway. *Introduction to VLSI Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1979.

[43] C. Mead and I. Sutherland. Microelectronics and computer science. *Scientific American*, 237(2):210–228, Sept. 1977.

[44] A. Moors, T. Rompf, P. Haller, and M. Odersky. Scala-virtualized. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, PEPM '12, pages 117–120, New York, NY, USA, 2012. ACM.

[45] G. Nadathur and D. Miller. An overview of lambda-prolog. In *ICLP/SLP*, pages 810–827, 1988.

[46] M. Odersky. *The Scala Language Specification*. École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland, 2011.

[47] E. Park, S. Kulkarni, and J. Cavazos. An evaluation of different modeling techniques for iterative compilation. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, CASES '11, pages 65–74, New York, NY, USA, 2011. ACM.

[48] M. Pericàs, X. Martorell, and Y. Etsion. Implementation of a hierarchical n-body simulator using the ompss programming model. In *Proceedings of the first workshop on Irregular applications: architectures and algorithm*, IAAA '11, pages 23–30, New York, NY, USA, 2011. ACM.

[49] N. Pouillard. Nameless, painless. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 320–332, New York, NY, USA, 2011. ACM.

[50] B. Pradelle, P. Clauss, and V. Loechner. Adaptive runtime selection of parallel schedules in the polytope model. In *Proceedings of the 19th High Performance Computing Symposia*, HPC '11, pages 81–88, San Diego, CA, USA, 2011. Society for Computer Simulation International.

[51] Y. Pu, R. Bodik, and S. Srivastava. Synthesis of first-order dynamic programming algorithms. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 83–98, New York, NY, USA, 2011. ACM.

[52] M. Püschel. Compiling math to fast code. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, PEPM '12, pages 1–2, New York, NY, USA, 2012. ACM.

[53] J. Ributzka, Y. Hayashi, J. B. Manzano, and G. R. Gao. The elephant and the mice: the role of non-strict fine-grain synchronization for modern many-core architectures. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 338–347, New York, NY, USA, 2011. ACM.

[54] G. Roşu and A. Ştefănescu. Matching logic: a new program verification approach (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 868–871, New York, NY, USA, 2011. ACM.

[55] D. Selwood. The inmos legacy, Aug. 2007.

[56] A. Shali and W. R. Cook. Hybrid partial evaluation. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 375–390, New York, NY, USA, 2011. ACM.

[57] T. Sheard. Languages of the future. *SIGPLAN Not.*, 39(12):119–132, Dec. 2004.

[58] A. Stump. Directly reflective meta-programming. *Higher Order Symbol. Comput.*, 22(2):115–144, June 2009.

[59] I. Sutherland. The sequential prison. *SIGPLAN Not.*, 46(10):1–2, Oct. 2011.

[60] K. Tian, E. Zhang, and X. Shen. A step towards transparent integration of input-consciousness into dynamic program optimizations. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 445–462, New York, NY, USA, 2011. ACM.

[61] V. Ureche, T. Rompf, A. Sujeeth, H. Chafi, and M. Odersky. Stagedsac: a case study in performance-oriented dsl development. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, PEPM '12, pages 73–82, New York, NY, USA, 2012. ACM.

[62] M. Vaziri, R. Fuhrer, and E. Duesterwald. Open language implementation. In *Proceedings of the 4th International Workshop on Multicore Software Engineering*, IWMSE '11, pages 41–42, New York, NY, USA, 2011. ACM.

[63] P. Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 45–52, New York, NY, USA, 1984. ACM.

[64] D. Wagelaar, M. Tisi, J. Cabot, and F. Jouault. Towards a general composition semantics for rule-based model transformation. In *Proceedings of the 14th international conference on Model driven engineering languages and systems*, MODELS'11, pages 623–637, Berlin, Heidelberg, 2011. Springer-Verlag.

[65] Wikipedia. Hol (proof assistant) — wikipedia, the free encyclopedia, 2012. [Online; accessed 12-December-2012].

[66] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine. Active pebbles: parallel programming for data-driven applications. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 235–244, New York, NY, USA, 2011. ACM.

[67] L. Zhao, G. Li, B. De Sutter, and J. Regehr. Armor: fully verified software fault isolation. In *Proceedings of the ninth ACM international conference on Embedded software*, EMSOFT '11, pages 289–298, New York, NY, USA, 2011. ACM.

[68] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. Using a "codelet" program execution model for exascale machines: position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 64–69, New York, NY, USA, 2011. ACM.