Silmaril, A Functional Language for Distributed Parallel Evaluation

David M. Rogers

Abstract
Extending the λΠ calculus with first-order directories and loadable base and primitive types leads to a modern high-level language for implicit parallel distributed computing. Every program corresponds to a syntax term, and can be transparently stored/read/referenced from a distributed hash-table as a parsed syntax tree. The built-in namespace and revision control systems facilitate collaborative coding, language extensibility, JIT-compilation, and distributed parallel evaluation via work-stealing. Formal program syntax, evaluation semantics and typing rules are presented, along with examples of the networked revision-control system and introspective capabilities. Further work on the formal properties of the language will greatly improve this preliminary implementation.

Keywords
1. Introduction
The structure of the lambda-calculus is ideal for composing large programs at a high level. Lower-level program constructs such as loops and variable mutations are inherently imperative and more difficult to implement efficiently using functional syntax. Despite their differences, these two camps are growing increasingly closer.[54] Imperative operations can be specified functionally using monads.[56] Variable mutation is discouraged, and function-passing interfaces are encouraged in modern program design.[11]

Applications of functional language constructs in high-performance and distributed computing are just beginning to be recognized. All three of the Darpa high-productivity grand challenge languages, X10, Fortress, and Chapel,[16] included some language features for building models using their mathematical properties.

Sil incorporates three novel design features – first-class directories, automatic object serialization and versioning, and loadable extension types and primitives. Together, these allow it to be extended with efficient low-level code while relieving the programmer of the burden of manual serialization, storage, scheduling, and tracking of objects on parallel computers. This paper details the motivations, methods and outcomes of this approach.

Practically, functional programs are executed efficiently by using machine-native basic datatypes and primitive operations. In the Glasgow Haskell compiler,[36] these imperative-style primitives can be compiled using LLVM[30] at run-time. In support of this, most commonly used functional language implementations immediately compile function expressions into opaque objects representing executable code and a closure capturing the current execution environment.

Unfortunately, this practice snatches away the program text, removing it from program-level introspection. The practical consequence is that altering the execution flow of the compiler to implement new features becomes nearly impossible. As an example, consider the Idris language.[9] Although implemented in Haskell, which itself includes LLVM support, its execution path requires its own LLVM stack. Ideally, a high-level language such as Haskell would make a low-level internal representation available as a target for new languages. Significant progress toward such an interface has been made recently with the introduction of template Haskell[46]. Earlier work was limited to parser-level splicing operations[51] for multi-stage compilation, or partially redundant mini-language implementations using GADTs[59] for implementing domain-specific languages.

This strategy can be seen, for example, in the FFTW compiler[18] for polynomial evaluation, written in Ocamli.[58] The domain-specific transformations for polynomials work on syntax trees composed of symbolic addition, subtraction, negation and multiplication operations. There are related symbolic projects, such as the Spiral[45] program for compiling signal transformations produced by a the domain-aware Gap[22] language.

These mini-languages often do not include support for function abstraction and application, limiting their applicability and expressiveness. In contrast, the original Lisp language[49] included meta-circular facilities for run-time inspection and transformation of parsed Lisp code. This enabled translation of higher-level constructs into lower-level ones. This translation by successively removing high-level constructs is the basis for compilation. Modern lisp compilers no longer permit deconstruction of lambda (function) terms. Instead, macro expansion on the function text must be used to accomplish the same effect. Macro expansion introduces a host of difficulties, including expansion order, the treatment of bound variable names, and the inability to partially evaluate expressions before macro application.

Modern functional languages such as Haskell and Scheme[4, 36] have a flexible-enough syntax that it is possible to define domain-specific constructs within them. If this is done as a direct translation, the domain program will run the entire computation. If implemented as a symbolic syntax tree, domain-specific optimizations can be done. However, this makes it impossible to create functions and other objects in the host syntax without introducing a new Lambda and Apply (and all other operations) in the domain syntax.

This latter conundrum seems to require that all DSLs that are syntactically larger than the host language must implement their own compiler. This is not really the case, since techniques such as multi-stage compilation exist. These aim to selectively run computations during generation of a DSL, so that applications can be
embedded into the host language correctly. Another solution exists. If an entire compiled expression were available for introspection inside the host language, symbolic domain-specific operations could be ‘written-out.’ Successive runtime re-writing steps for removing each domain-specific level would be expensive, but essentially equivalent to a compilation step in more traditional multi-stage compilation.

There are many examples of numerical libraries that create specialized code either before or during execution, including the FEniCS project’s compilation of weak forms for partial differential equations,[32, 33] the Theano[7] library for combining array operations in python, and the distributed operations in S3D,[42] PETsc,[6] ScaLapack[15] Atlas[57] and Magma[5] libraries. Some of these optimizations are enabled through testing of multiple rewrite rules using run-time profiling. Although the latter projects mentioned have an inherently imperative design, their API’s are evolving toward more functional, mathematical expression.

In an intrinsically related, but traditionally separate domain are the JIT-compilation operations invoked during execution of virtual machine languages. Although these transformations see a very syntactically restricted language, they form the indispensable last step in the code transformation hierarchy. Opening up the syntax tree of a running program to introspection and allowing multiple re-write and evaluation strategies is the key leap required to enable all these developments.

This paper presents the language syntax, with separate sections devoted to directory types and to value-matching operators. Directories formalize name scope through module naming environments, and create a path for inter-operable code objects accessible through globally addressable, persistent, hash-based storage. This frees the source code to be evaluated in the cloud, and also allows the automated creation of specialized compiled code through extension datatypes and primitives. The next section presents parser-expression grammars as a central enabling technique for deconstructing program values. These deconstructions have been implemented using just two primitive operations. Finally, some conclusions are drawn on the current status and future of this work.

2. Language Specification

Dependently-typed languages require have a rich typing system for expressing high-level concepts. Efficiency also demands binary representations for language base types and compiled primitive operations (delta-reductions) over those types.

With these in mind, we define the language (Fig. 1) based on the standard $\lambda\Pi$ calculus, with the addition of extensible base and primitive types. These extensions greatly improve the flexibility and efficiency of the language at the expense of requiring the implementation to take ‘on faith’ the stated type of each primitive operation. Of course, the result produced after applying each primitive can be verified.

A similar type introduction occurs when defining new data constructors (type Sym). Each data constructor creates an abstract data type. The definition of natural numbers can, for instance, be implemented using three symbols, defined using the syntax, Nat : * , Z : Nat, and Succ : Nat → Nat. Type-checking applications of symbols is exactly analogous to checking function applications.

A Sil program is a term in this syntax. Top-level directories enable this term $\iff$ program correspondence by removing any requirements for ancillary data in the program specification. The evaluation (and error handling) semantics thus makes use only of expressions in the language. This is an essential part of the user experience for the language, since all aspects of programming and evaluation can be done from within the same language interpreter.

Each base type is globally unique, with a unique implied binary data format. Defining a new base type can be done by providing display and serialization functions. Base constants are each associated with a base type, and hold a block of binary data. They are instantiated and operated on in the program through the use of user-defined primitive functions.

Every primitive function has a unique name and is understood at the language level through its type. They are reduced using compiled code, allowing them to manipulate the binary form of base constants. For simplicity, only single-argument primitives are listed in the syntax of Fig. 1. In practice, the language transforms all primitives into a series of $\lambda$-functions that collect (dependent) arguments into a tuple, which is sent to the primitive when a delta-reduction is required. The semantics for introducing new base types and primitives is explained in Sec. 2.3.

There are several pre-defined, (binary) base constants. These include booleans, integers, floating point numbers, strings, and their types. Lambda functions and applications form the prototypical members of the value and term classes. User-defined functions (primitives), the type of functions (\(\Omega\)), base constants and their types also belong in the class of values.

Directories, or lists of bound terms (aka. modules, records, tables), are also made into first-order values in this language. Each directory contains a list of included directories, and a list of (mutually recursive) variable bindings. The lookup (Elem) operation searches only the names directly bound by the directory. Directories also function as contexts, since variable names appearing inside terms in a directory are resolved by checking first the directory’s own variable bindings, then variables directly bound in the ordered list of included directories. Includes of includes are not checked. Directory hierarchies are created by listing parent directories in the inclusion list. While allowing arbitrary inclusions, this ordering fixes the recursive include problem by limiting the search depth for any given context to \(O(1)\). Directory types are defined following Refs [14, 21] (explained further in Sec. 3).

Allowing symbolic constants Sym that stand in for things like strings or real numbers means that some applications cannot be reduced. To represent evaluated programs, we define a syntax for neutral terms. Values thus describe the syntax of all possible evaluated programs.

The language makes an attempt at graceful, functional error handling through the syntax of poison values. Poison values halt the progress of program evaluation, resulting in stuck applications. Unlike unevaluated terms, both sides of stuck applications can be completely evaluated. Refusing to substitute poison terms is also important for consistent behavior when matching under lambda terms. There, symbolic ‘doppleganger’ constants are created to take the place of the bound arguments. Errors during program parsing or typing produce error terms, which can show multiple errors at once and their relative locations in the code. Terms with illegal types are thus turned into poison values which permit well-typed use in the language.

The addition of directories and user-defined bases and primitives enables several novel uses of the language. Notably, the directory terms provide a means for organizing both the source code and results of entire projects. This strategy has proven highly successful in the Lua language.[23] Operations within this framework provide their own filesystem without the overhead of manual serialization and parsing. Persistent storage for every program is implemented through mandatory serialization functions associated to each base type. Section 4 will show how a single primitive that destructures neutral and stuck applications adds meta-circular properties to the language.
Imperative and sequential updating expressions can be built using the special types[25, 26]

There is a special mechanism for declaring primitive operations. The assignment, '#bits = 54 * 7 - 4' assigns 374 copied everywhere it is placed. It can be altered using names pre-appended to parent directories in the initial inclusion list of directories. This ensures that no committed directory will refer to a symbolic doppleganger.

Figure 1. Sil expression syntax
2.2 Bootstrap Primitives

The language is bootstrapped through adding the following primitives to the standard environment:

- **store**: \(?((a : *) \to string) \to a \to IO(a)\)
  Store Ast as a named object in the current commit. These are created by the parser from the syntax “\text{NAME} = expr”.

- **commit**: \(\{\} \to IO(string)\)
  Commit a directory to permanent storage and return its hash string.

- **open**: \(string \to \{\}\)
  This returns a commit from a hex-encoded hash value.

- **Base**: \(string \to s_{0}\)
  Used to declare types when defining primitives

- **Prim**: \(?((a : *) \to string) \to a\)
  This is used to load an initial reference to a primitive function.

Hole-binding constructs can meaningfully coexist with primitive operations. First, IO actions can be carried out for objects with the type \(IO(a)\), where \(a\) is any type. Hole-binding contexts that get between the top-level term and \(IO\) can be (correctly) shifted to the right of applying \(IO\). On the other hand, primitive functions that have a hole-bound variable as argument will not be able to look up their values. This is the same treatment as lambda-bound variables (where the lambda has not found a corresponding application).

It should be noted both adding to the current commit and storing directories are considered as IO actions, whereas loading committed directories are considered as pure code. The process of writing code implies variable mutation, which has to take place through a sequential, imperative process. On the other hand, reading immutable data values is part of a pure, functional process.

Thus, all references to other code take the form of linking between (globally addressable) libraries. Loading primitives and base types make use of loading directories, and are considered as pure. This separation consciously chooses to accept some level of indeterminacy in resolving library references, since code may be lost or unavailable outside the scope of a particular organization. Nevertheless, the issue of finding all required dependencies is not new to programming languages, whose purpose is merely to specify how to produce a value.

Primitive types can be defined both by the load\_base type constructor as well as through a \texttt{typeof} function. Because the language takes care of type-checking expressions, the typeof function is implemented as the trivial \texttt{typeof}\((x : *) a = a\). All other language constructions can be accessed through the parsing process.

2.3 Language Extensions and Embedding

Sil uses directories to provide long-term distributed storage and addressing. Because every term in Sil can be serialized, we can produce essentially unique storage addresses by hashing the serialized form of a directory. References to external libraries are encoded through this hash, providing a secure information distribution system.

Once a chunk of code is committed, it has a universal address that can be used to load it as a named object or to extend the unqualified namespace of any given working directory. This provides a natural mechanism for linking with external libraries. Any observer with the ability to load both the new project and the linked project can reference either or both. Additionally, any observer can also evaluate the combination to a compiled form which makes use of optimized code and data structures.

Language extensions are used to provide optimized primitive functions and data structures. They also work through the directory store/retrieve mechanism. Committing a directory with the type:

\{
  \text{API} = string;
  \text{name} = string;
  \text{code} = string;
\}

is all that is required to create a new base type. The API describes how the language should interpret the base type. Currently, the only API is described here, and is denoted by a blank string for API.

The code object contains C source for functions with the following prototypes.

```c
char *show(void *e);
char *serialize(int *len, void *e);
Ast *parse(Allocator *a, size_t len, void *buf);
```

 Arbitrarily complex data structures can be defined as base data types, enabling any code fragment to run at machine-level. Sharing large data objects can result in a large win. The execution system only copies these base objects during (de)serialization, and when sending them as input to "ST" actions.

Similarly, primitive functions can be added from a commit with the type,

\{
  \text{API} = string;
  \text{name} = string;
  \text{extn} = int;
  ni = int; --number of ignored (head) type args
  n = int; --number of unignored args
  ptype = *;
  reduce = string;
\}

These are instantiated into values using load\_prim, described above. The returned primitive is actually a series of \(ni + n\) lambda-functions which have been applied to \(ni\) hole values and reference the remaining \(n\) in the primitive’s argument list. The primitive is fully applied (and can be reduced) if all its the arguments are un-poisoned values. Primitives provide the only mechanism for working with base data types, including initially creating them from other values in Sil.

Since primitive functions are dependently typed, they also provide a powerful language extension mechanism. The if statement, for example, can be represented by the primitive,

\{
  \text{API} = "",
  \text{name} = "if/2",
  \text{extn} = 3,
  ni = 1,
  n = 3,
  ptype = ?((a : *) \to bool) \to a \to a \to a\}

and a suitable reduce function.

The reduce string contains a C function implementing the prototype,

```c
Ast *reduce(struct { void *st;
          Ast *com;
          Allocator *a; } *),
          struct Prim *);
```

For IO primitives, \texttt{extn} is 1, and \(st\) contains a pointer to a structure holding the current working environment. State-modifying primitives (\texttt{extn} = 2) are run sequentially, just as IO primitives, but sent an Ast as the mutable state. The \texttt{st} object is undefined for pure primitives (\texttt{extn} = 3). To permit parallel garbage collection, all memory allocation takes place using the allocator. The \texttt{Prim} structure links to the primitive definition in an internal table, and contains a list of \(n\) Ast pointers to the arguments.
Because the language is completely functional, ways to make it dovetail with a host language are not intuitive. The language has to be understood as a specification for creating values. Execution is the mapping of terms to values. At the inter-program level, this execution is asynchronous. There are thus two roles a host language can play. First, the host language can create Sil terms and wait for an external evaluator. Second, the host program can read terms, perform code transformations or evaluations of some part, then store the resulting terms. This second type of interaction is the most interesting, since it allows the possibility for a given host evaluator to selectively operate on programs based on their properties. A low-latency, high memory cluster may selectively carry out array computations, or an installation with a specialized software stack may exclusively reduce primitives involving external code requiring a large infrastructure.

3. First-Order Directories

Collections of named objects form the basis for modules collecting groups of code, typed interfaces, and nested namespaces. By making these first-order, we can get ML’s parameterized modules[40] for free, by creating a module in the body of a function. Moreover, these modules present the functional equivalent of an object-oriented programming interface. Finally, modules provide a ms for breaking up large programs into smaller compilation units. Other works have explored the use of modules before as well,[20] including the usefulness of module dependency graphs in package management[28].

The pervasive use of first-class directories in Sil has several similarities with the Lua language,[23, 24] There, tables of named values are widely used to describe engineering attributes, window layout information, and code for scripted actors in games. Introspective listing of directory items is commonly used during data construction to fill in derived and default values. A data constructor performing such activities, e.g.

\[ \text{Window}\{x = 100, y = 200, \text{color} = \text{“red”}\}, \]

would have exactly the same syntax in Sil. The formal semantics and strict functional interpretation differ from Lua. The largest difference is in the strict typing interface for base datatypes and primitive operations. Code purity allows the Sil interpreter to work in a distributed, asynchronous parallel fashion. It also has the benefit of re-using shared values – of great use for resuming partially failed computations. In addition, static typing allows computations to abort early, alerting developers to issues earlier. The sacrifice required for these properties is a sometimes more cryptic syntax.

The typing rules for the basic syntax of \(\lambda\) are well-known,[9, 34] Figure 2 presents additional rules for typing directories and their elimination (through Elem). The introduction rule states that the type of a directory containing terms is a directory containing types associated to the same names. The formation rule terminates the directory type-checking hierarchy by stating that directories containing only valid types are valid types.

Elem operations require a corresponding directory entry to be present. The type of a directory can thus be inferred inside a context by unifying the types of all Elem operations. Conversely, the reduction rule ensures that directories with ancillary elements are valid. The type of a directory can therefore be checked against a specification by ensuring that it contains at least two required elements.

It is significant that the reduction rule only removes key/value pairs from the type of a directory, not the directory itself. This insures that a directory containing both types and values, such as \(\{a = \text{int}, a = 1\}\), has a type that can be passed to functions requiring any of the four possible inclusion/exclusion combinations of \(\{a = *, a = \text{int}\}\), but can never unify with *. In other words, the directory \(\{x = *\}\) is not equivalent to * because only the types narrow, not the values.

The language also implements a similar set of typing rules for tuples, which act like directories with elements that are consecutively numbered from zero rather than named. Like directories, tuple types can be narrowed, so all tuples unify with typeof (\{} \(= \text{nil}\). This behavior shares some similarity with Julia’s record and tuple data types,[8] but retains the invariant typeof (typeof (\{} \(= *\).

3.1 Directory Evaluation

Evaluation rules have to be extended for directory types. Using the analogy between directories and a series of let-binders, these are straightforward. Directories do not need to be evaluated until an Elem operation is applied, which then transforms the expression into a let-bound application.

\[
D.x \rightarrow V[y_i := D.y_i, z_j := E[j], z_j]
\]

where

\[
D = \{\{E\}, x = M, y_i = N_i\}
\]

and

\[
D \vdash M \Rightarrow V
\]

First, the value bound to \(x\) is evaluated to \(V\) inside the directory scope (denoted \(M \Rightarrow V\)). Next, all remaining identifiers inside \(V\) are re-associated to the correct reference for the parent space. These identifiers include both names, \(y_i\), directly bound inside \(D\) as well as names, \(z_j\), that resolve to included directories, \(E_i \in \Lambda\).

For directories that contain simple blocks of assignments, as in \(x = y * (y + z), r = x * (x - 2)\), the evaluation step mimicks traditional function execution in block-structured languages. The re-naming step is needed when functions or sub-directories with references are requested. Retaining the reference to the original structures permits a lazy evaluation strategy for directory-bound names. A single global cache can be used to store evaluated forms for each directory, and commonly accessed functions can be singled out for optimization.

3.2 Distributed Computing

One way of organizing distributed evaluation of a module hierarchy is via a distributed work-stealing scheme,[19, 48, 52] Using the code hash as a key, both the unevaluated source and evaluated program code of each directory can be stored in distributed a key-value store.[31] On encountering a source code reference, an evaluator can check for a mapping to its evaluated version. Compute nodes can find the address of an unevaluated module from a shared bag of pending tasks. Evaluation may require dependencies, which can be added back to the bag. The use of hash-derived names resembles the practice of strong naming in the .NET framework’s global assembly cache.[38] Some distributed key-value store implementations (e.g. RiaK[3]) support map-reduce operations targeted at searching and commit callbacks that would be appropriate for queueing tasks.

This model is inherently asynchronous and parallel. A replicated key-value store would lend it two types of redundancy. First, copies of each source fragment would be stored in multiple locations. Second, should part of a running evaluation be interrupted it can be trivially resumed by any other compute node. This mechanism can be utilized to safely remove results occupying much more storage space than their source, since the former can be re-created from the latter. Just like makefile-based source compilation, computations can fail gracefully by retaining valid intermediate results. These last two properties can be especially important for data pro-
cessing tasks which require reproducibility and efficient re-use of intermediate results.

The immutability and Merkle-tree structure of code stored in this way also guards against various versions of DLL and JAR hell. In particular, no dependencies are replaced. Even different parts of the same project can depend on incompatible versions of another library. This type of issue is rampant in system-level package managers, and even Haskell’s Cabal is affected by it.

At the same time, the granularity of storing individual directories should be sufficient to ensure that a large quantity of core library code is shared between projects and rarely requires updating.

### 3.3 Potential Issues

One interesting case is the possibility that two development teams create a mutual reference to one another’s code. This could be introduced when developer A produces a package with feature Y, that uses feature X from developer B. Developer B notices that feature Y is superior, and re-codes feature X to work by calling feature Y in their next release. When Developer A releases an update, and decides to depend on the updated version of developer B’s project, B’s current project still references the last version of A’s project, creating a ping-pong call sequence leading all the way back to the first version of feature X. Each pair of version updates introduces two new references.

This type of problem is difficult to imagine in the current, ad-hoc, distribution hierarchy. This is because it is impractical to install a library that depends on a package that depends on an earlier version of your library. It would be irreconcilable without developer collaboration. The problem case above is symptomatic of a truly distributed infrastructure, where any external code can be immediately used as a library.

One immediate remedy is to check, on considering using a new external library, that the transitive closure of its code references does not lead back to any code in the current project. Another is to check for updated releases of dependency projects at the development, release or evaluation stages. The universal naming properties, cryptographic signatures, and strong typing system for Sil greatly facilitate finding, validating, compatibility checking and installation of updated dependency versions.

### 4. Match Expressions

Most functional languages include a mechanism for defining new data constructors and parsing them through matching. A simple scheme for this[9, 35] uses a set of tags, $K_i$, each expecting a set number of terms. Match clauses take the form $K x_1 \ldots x_n \rightarrow M$. It is easy to see how matching could be extended to the subset of neutral terms and values including only symbolic constants, base constants, base types, and primitives. If, in addition, there were a mechanism for dealing with values that express binding constructs (functions, directories and their types), then the language would have many of the favorable properties of meta-circular languages.

Deconstruction of function types was one of the original capabilities of LISP, where it can still be accomplished textually using simple quoting. Implementing such facilities in a typed programming language such as Haskell is much more difficult, owing to its principalled structural representation. One class of approaches uses an extension of the unification algorithm.[39] Rather than include match clauses into the language itself, we explore here the use of dependently typed match functions with the semantics of parser-expression grammars. A parser-expression grammar (PEG) consists of an ordered sequence of tests. Each test checks the next input token and, on success, returns the remaining stream and a value based on the token. On failure, the next test in the sequence is performed. When no tests remain, the whole expression fails. Because tests in a PEG can be nested, PEGs are capable of parsing recursive grammars. Because rules consist of test functions, they can be named and composed.

Match types are implemented using continuations (Fig. 3). Matches operate only on whole values, and do not return unparsed input. Their type is a slight modification of a pattern that passes both success and failure callbacks to each parse attempt. Implemented using this success/failure method, each attempt either returns its success continuation applied to current parsed value or returns the default, failure result.

The scheme can be modified to combine the two continuations into a single continuation over the option (Maybe) monad. However, this solution is not entirely satisfactory, as the destructuring implementation would itself require destructuring “Just” values. Instead, the continuation and maybe monads have been combined into a dependently-typed conditional destructuring operation with the type MaybeF.

The possib function creates MaybeF-s using a polymorphic if function.

\[
\text{select} : (e : \text{bool}) \rightarrow a \rightarrow b \rightarrow \text{if e then a else b}
\]

Like if, it is implemented internally by a primitive. All the case statements for the continuations, a, can be written in terms of possib.

Expression matching is invoked at the top-level through runMatch. The initial continuation returns the match result on success and the default value, d, on failure. The initial input is formed by a. Ordered alternative tests can be combined by replacing the failure case with the second match attempt. Sequential matches are added to the success case.

The sequencing combinator matches applications of the form $n \ \ V$. All sequences of applications have symbolic constants at their head. Because matches operate on values, the sequencing construct has to be right-associative – matching $n \ \ V$ by matching $n$ and then $V$. Unlike the ordered choice combinator, $\#$, matches applications, and therefore requires a primitive function (unApply) for its implementation. This semantics matches the LISP-style treatment of s-expressions as lists. The major difference here is that quoted expressions are typed, and created by storing values using symbolic data constructors.

Obviously, for the framework to be referentially transparent, each match attempt must start by evaluating its argument to a neutral term. Thus, matches can only succeed on values or neutral

---

**Figure 2.** Typing rules for directories

Form

\[
A_i : \forall i \neq j \in S \vdash A_j : \star
\]

\[
\text{Red } \forall T \subseteq S \vdash \Lambda : \{ x_i = A_i | i \in S \}
\]

Intro

\[
\Lambda \vdash x : \bar{A}
\]

Elim

\[
\Lambda : \{ x = A \}
\]

\[
\Lambda : x : A
\]
terms. It would not make sense in this framework to try and match
dent variable-argument types.

| MFmap f cont e = IF e then fun a -> cont true (f a) |
| MFmap : (a -> b) -> MaybeF r a -> MaybeF r b |

| returnC e = IF e then fun a cont -> cont true a |
| returnC : (e : bool) -> IF e then (type a -> MaybeF b a -> b) |

MaybeF r a = type (e : bool) -> (e : bool) -> IF e then fun a cont -> cont true a |

possib : b -> (a -> b) -> MaybeF b a |

possib b f = fun e -> IF e then f else b |

Match r a b = type a -> MaybeF r b |

runMatch : Match r a b -> r -> (b -> r) -> a -> r |

// : Match r a b -> Match r a b |

-- g // h = fun a s f -> g a s (h a s f) |

g // h = fun a cont -> g s |

where s false = h a cont |

s true c = cont true c |

infixl // 0 |

$$ : Match r a b -> (type b -> c) -> Match r a c |

g $$ f = fun a cont -> g a s |

where s false = cont false |

s true b = cont true (f b) |

infixl $$ 1 |

$$ : Match r (type v -> n') (type b -> c) -> Match r v b |

-- g @ h = unApply (fun n v cont -> g n s) |

g @ h = fun (n v) s f -> |

unApply (fun bc -> h v (fun b -> s (bc b))) f f |

where s false = cont false |

s true bc = h v (MFmap bc cont) |

infixl @ 2 |

**Figure 3.** Matching expressions using continuations and depend-
ent variable-argument types.

terms. It would not make sense in this framework to try and match
an unevaluated term of the form, \((\lambda x \rightarrow V) W\).

As the parse proceeds, the input is consumed and turned into a
match result. Common pattern matching constructs produce an
expression with bound variables as the result. This is achieved in
the present formalism by assuming the left-hand side of each match
(thus ultimately the production that matches the leading symbol)
produces a function that reads the right-hand values (or their match
results).[10] This sacrifices some of the composeability of PEG
parsers, but perfectly compliments destructuring via dependently
typed application shown in Section 4.1. The successive application
structure also leads to simple parsers.

A universal match is expressed by

mWild : Match r a a

mWild a = returnC true a.

It returns the previously parsed expression and moves the rest of
the input to the end of the parsed output. Conversely, mFail just returns
false through the continuation.

Equality tests form the simplest match constructs. The == operator is special in Sil, in that it implements equality over a subset of
neutral values with the semantics of Ref. [29]. The subset includes
all base values and terms representing types, where equal-
ity is well-founded. However, the current implementation is tem-
porarily incorrect in that it does not force evaluation under \(\lambda\), and
hence is not suitable for functions due to the possibility of \(\eta\) and
\(\beta\)-equivalence. Constructing a match for a specific int, string, sym-
bolic type etc. is simple.

mEq : a -> Match r a a

mEq i j = if i == j

then returnC true j

else returnC false.

Traditional string PEG parsers can be built by introducing a
sequencing operator for string Match objects that return the
unmatched string in the output.

| \(\text{pstring}\) r b = Match r string b (b, string, string) |
| \(\{+\}\) : \(\text{pstring} r b \rightarrow \text{pstring} r b \rightarrow \text{pstring} r b |

\((g \{+\} h)(s, b)\) cont = \(g (s, b) f\)

where \(f\) Nothing = cont Nothing

\(f\) Just (b, s, t) = h (t, b) (u s) |

\(u\) Nothing = cont Nothing |

\(u\) r Just (b, s, t) = cont(Just(b, str.join r s, t))|

The definition of a function can thus contain arbitrary string parsing
code for reading its arguments.

Matches act as functions, as their output can be re-mapped
using the $$ construct. This can be used to manipulate the input
expression and to bring the results of all sub-matches into compatible
types.

Traditional matching constructs are therefore implemented in
Sil through three basic primitives - an equality test function, a
higher-order if function, and an unAppLy function for deconstruc-
ting neutral terms. With the help of a standard primitive to decom-
pose substrings, the parser gains all the features of a traditional
PEG parser. While this direct implementation is horribly inefficient
for strings, alternative parsers can be used for actual text.

The extra work required for implementing matching using a
series of basic matching functions is balanced by the ability to
compose matching expressions. Compositional parsers solve many
language design problems, and can be used to substantially reduce
code size (redundancy).[2] One example of this compositional-
ity is de-structuring a type-level unary number into an integer, using a
recursive match,

NatElim : Match r Nat int

NatElim = mEq Zero $$ const 0 // (mEq Succ $$ (+1)) @ NatElim.

### 4.1 Symbol-free Matching Constructs

| TBox : (string -> 'kind) -> 'kind -> 'kind |
| TBox f r = type (s : string) -> f s -> r |

string_or_int s = if s == "String" then string else int

unBox : TBox string_or_int int

unBox s = select (s == "String") |

fun x -> str.length x |

fun i -> i |

unBox "String" "text" --> 4

unBox "Int" 42 --> 42

**Figure 4.** Self-unboxing dependent-type constructors.

The use of a type-level if statement in the definition of possible
in Fig. 3 shows the power of dependent typing. The sequence values
passed to a function can depend on the previous passed values.
MaybeF type used in Fig. 3 combines the 'Maybe' type with the 'maybe' deconstructor function. Figure 4 generalizes this idea to show that matching functions may not be strictly necessary at all. TBox constructs polymorphic boxed data types which contain their own deconstructor. A function wishing to unbox the data type can then dispatch on the argument type. This scheme has the drawback that compound boxed datatypes have to be passed as two successive values. This, in turn, requires defining more complicated returnC and fmap functions for these types.

### 4.2 Matching Binding Terms

An alternate motivation for this approach is to build meta-circular properties into the language. Because these constructs do not introduce additional names, any framework capable of matching arbitrary values in the syntax of Fig. 1 can also match the matching terms themselves. Functions, function types, and directories are complicated because they bind names. Primitive match objects for these values are required to make the language completely meta-circular. Since recursion in this syntax is introduced through names, this also introduces the problem of matching recursive expressions.

Matching the inner side of a binding expression will create free variables. For instance, attempting to match the expression

$$\lambda x \to 10 \ast x$$

requires both a primitive to de-structure the lambda and a match object to recognize the body of the function, $10 \ast x$. Its specific form is only uncovered when applying the match to a lambda-value. If the inner match expression is a wildcard match, its parse result will reference the free variable, $x$. The inner match expression may also want to recognize the bound variable, $x$, for example to carry out a substitution.

Handling both these cases requires the lambda-matching primitive to wrap the execution of the inner match. On entrance, a unique symbolic name is substituted for the bound variable in the function body. The inner match expression gets the value of this symbol, and is invoked to produce its match result (or fail). On successful exit from matching the lambda primitive, all instances of the unique symbol returned in the parse are substituted by a default value. This semantics ensures that only closed terms are produced as parse results of lambda matches.

The lambda match then requires a default binding and a function taking a unique symbol and producing a subexpression match. The first argument to the match function is a function that will produce the default binding. The second is obviously the match construct, which works on the unique symbol and the function body. Instances of the unique symbol remaining in that match function's output, $C$, are replaced by the default.

Lambda values can be deconstructed using

$$\text{mLambda} \quad : ?x : A \to ((\Pi A \to B) x) \to \text{Match } r \ A B x \ C x \ \
\text{match } r \ (\Pi A \to B(x)) \ C$$

This is implemented similarly to unApply using a 1-step deconstructor,

$$\text{unLambda} \quad : ?x : A \to (\Pi A \to B x) \ \
(\Pi A \to B x \to C x) \to ((\Pi A \to B x) \to C x)$$

The semantics of this de-structuring is very similar to the inductive function elimination rule, funsplit($f$, $d$), introduced in Ref. 41. There, $f$ has type $\Pi(x : A) \to B(x)$ and $d$ receives the function body with type $?x : A \to B(x)$. The de-constructor, $d$, then produces a value whose type can depend on the function body, $?g : (w : A) \to B(w) \to C(y)$. For sanity, we have chosen to let the match result type depend on the function’s input instead.

The hole-binding of the actual argument $x : A$ is required to match functions with dependent types. This hole binding can be removed if the matched function’s type is not dependent, i.e. $B x = \text{int}$, etc. Otherwise, the hole-binding is carried through to the matching function. A match on the identity function is one example, $\text{mId} = \text{mLambda id} \ (\text{mPairEq} \ \text{success}_f)$. This works only on functions of type $a \to a$, and uses another match function, $\text{mPairEq}$, to decide if the elements of a 2-tuple are identical. If the success function is the identity, the match result will be the unique symbol, which will be replaced with the original function argument during the unApply wrapper, re-constructing the identity.

In addition to defining macro-like constructs, another use for this matching facility is carrying out automatic differentiation for applications in numerical simulation.[27, 50] Because of the constrained introduction of unique symbols, free variable capture due to non-hygienic use of named symbols is not an issue.[47]

It is straightforward to use this same idiom to create match objects for $\Pi$ values. Lambda and $\Pi$ values share the interesting similarity in that both represent typed binding constructs. It is tempting to speculate on a type theory containing a closure similar to that used for directories here.

$$\lambda (x : A) \to b : \lambda (x : A) \to B \ \
\lambda (x : A) \to B : x!$$

where $b$ is a value, and $B : \ast$ is its type. Both bind the variable $x : A$, but the type of $B$ is uninteresting and does not bind $x$, and hence assuming any function that returns a type is itself a type is not completely unfounded.

Match objects suitable for deconstructing directories are beyond the scope of this work. Despite their importance in Lua, automatically typed directories are relatively new to functional programming. A limited form of deconstruction can be done by testing whether an input directory has a given type (implying it contains some requested elements). This could be used to implement type-level dynamic dispatch. One approach to complete listing is to render the directory as an association list. However, a consistent solution will require careful consideration of the typing characteristics of directories when moving between named variables and strings. For most cases, the ‘Elem’ primitive, which performs directory lookup, is sufficient.

### 5. Conclusions

This paper has presented an extensible framework for implicitly parallel, distributed compilation and evaluation. Including directory hierarchies into serializable functional programs makes a powerful combination. All data objects can be transferred and used in their parsed form, and full program introspection is available everywhere. Since compiled primitive operations can be generated on-the-fly, this scheme incorporates a compiler-compiler, domain-specific language evaluation, and bytecode into one. This unification is a key requirement for creating self-profiling algorithms.[18, 42, 53, 57]

Distributed evaluation presents a large range of possibilities, including automatic production of multiple compiled formats, co-location of data and computations, and resource specialization to execute a particular subset of applications. These behaviors are fundamentally enabled by a standard source distribution format and naming scheme. Cooperative evaluation activity can be driven by market-based resource management.[13]

The convergence of functional and imperative design patterns has lead to a remarkable consensus.[11, 26, 55] Pure functional procedures neatly encapsulate computations and have desirable optimization and transferrability properties. Sequential updates require
imperative operations, but can again be composed as a series of
monadic operations within a larger functional framework. Church’s
lambda-calculus extended with dependent types therefore emerges
as a natural language for organizing high-level calculations at a sys-
tems level.

Program parsing techniques provide a flexible and composable
path to implement de-constructors without any additional language
construct. To the author’s knowledge, its meta-circular application
to carry out symbolic differentiation represents a first in the history
of programming languages.

There are some drawbacks to back-tracking parsers such as
PEGs, including exponential worst-case complexity unless inter-
mediate parses are stored using a packrat strategy. Creating a re-
cursive data-structure representing the complete parse is one quick
solution, but requires lazy evaluation.[17] Another possibility is ex-
tending the PEG construct to include a memoization monad, or
switching to LL* style parsers.[43] There are also interesting so-
cial questions, such as the storage and distribution of a large body
of shared, immutable code, and the optimal way to encourage large-
scale distributed evaluation.

Future work is needed to address strong normalization and ter-
mination guarantees that can be provided by the type system.[9]
These can be used to strengthen confidence in parallel evaluation
strategies and garbage collection efficiency. Code memoization tech-
niques can also be used to determine common computations that
operate on simple datatypes as targets for optimization.

References

http://www.aldeivblogaday.com/2012/04/26/functional-pro-
gramming-in-c/.
[4] B. Chang and M. Felleisen. The call-by-need lambda calculus, revis-
it. In 21st European Symposium on Programming, volume 7211 of
Lecture Notes in Computer Science, pages 128–147. Springer Berlin
execution environment for clusters. In B. Falsafi and M. Lauter,
editors, 4th International Workshop, CANPC, volume 1797 of Lecture


