Abstract—We present a concurrent programming language Cilk-5 in terms of semantics and implementation [1]. We show how it relies on both the compiler and the work stealing scheduler to produce efficient execution schedules of programs, and give a theoretical analysis of its performance. We then introduce the concept of dataflow programming [9] in which programs yield the same observable results regardless of a particular execution schedule. This paradigm is useful for parallel programming, since it ensures that programs exhibit identical behaviour independent of the platform they are deployed on. Finally, we describe a method for embedding domain-specific languages in host languages using the typechecker of the host language to ensure typing rules are followed [7]. This technique allows using the staged representation of a program to compile it to different languages depending on the target platform.

Index Terms—Work stealing, Cilk-5, dataflow, single-assignment variables, deterministic concurrency, staging, embedding

I. INTRODUCTION

ONE important property of programs is observable determinism. Given the same set of inputs the program produces the same set of outputs regardless of the execution schedule. When extended with concurrency primitives imperative programming does not have this property. Programs written in this model are subject to data races and may produce different results on different runs. This is undesired in practice, since a program that’s been implemented and tested on one platform may exhibit unexpected behaviour when deployed on some other platform. Behaviour of a concurrent program depends on the execution schedule and different platforms tend to enforce different execution schedules for concurrent programs. This can lead to bugs that are not easy to reproduce, and thus not easy to fix.

Declarative concurrency is a programming model which ensures that programs written in it are observably deterministic. Different execution schedules may exist for these programs, but they always produce the same results given the same set of inputs. We show one declarative concurrent model as described in [9].

A programming model whose core constructs ensure determinism is provided by the Cilk-5 programming language [1]. This programming language relies on the spawn/sync (also known as fork/join) constructs which yield programs with fully strict multithreaded computations. It is a superset of the C language. We show the details of Cilk-5 and the implementation of its compiler and the runtime system. We describe work stealing scheduling and how it’s implemented in Cilk-5.

There exists a variety of different parallel platforms from multicore processors to supercomputers and clusters. The same concurrent program shouldn’t have to be written for each of these platforms anew. Instead, a program should be written once and efficiently executable on different platforms. We present staging [7] as means of obtaining program representation in a typesafe manner. The staged representation can be used to produce different code of the program for different target platforms.

II. CILK-5 MULTITHREADED LANGUAGE

Frigo, Leiserson and Randall [1] have implemented Cilk-5, a fifth version of the multithreaded language Cilk, which is a parallel extension of C. Cilk is a general purpose programming language which extends C by introducing keywords for parallel control. Cilk-5 uses work stealing scheduling algorithm to create threads, but unlike previous versions, both the compiler and the runtime system participate in the scheduling policy. We first describe the Cilk-5 language and its semantics, then give a short analysis of scheduling using work stealing and finally describe how the compiler and the runtime system work.
A. Cilk-5 Semantics

Cilk-5 extends the C language by adding five keywords. First of all, any method can be annotated with the keyword cilkn, meaning the method can be invoked in parallel. To actually run the method in parallel, an invocation of a method M should be prefixed with the keyword spawn. This runs M in parallel with the method it was called from, until a sync invocation is reached. At that point the computation stalls until all the methods spawned from the current method return. We illustrate this with the following simple Cilk program that calculates Fibonacci numbers:

```c
#include <stdio.h>

int fib(int n) {
    if (n < 2) return n;
    else {
        int x, y;
        x = spawn fib(n - 1);
        y = spawn fib(n - 2);
        sync
        return x + y;
    }
}

int main(int argc, char *argv[]) {
    sync = spawn fib(40);
    printf("Result: %d\n", result);
    return 0;
}
```

Both methods have a cilkn annotation, meaning they are Cilk procedures, the analogues of C functions. The main procedure spawns a new instance of procedure fib which runs in parallel and then stalls when it reaches sync. The fib procedure either returns a result directly if its argument is small enough, or invokes two recursive instances in parallel and waits for them to finish at which point it can return a result. Once a spawned procedure returns, its return value is integrated into its parent procedure’s frame. Normally, spawning a Cilk procedure must occur as a separate statement. To use the return value in a more complex way, it’s possible to spawning a Cilk procedure must occur as a separate statement.

Work stealing [1] [2] [3] [4] is an efficient technique for scheduling multithreaded computations among multiple processors. In scheduling a multithreaded computation by work stealing each processor is assigned some number of threads. When a processor completes its work and becomes underutilized, it attempts to steal threads from other processors. Each processor stores threads assigned to it in a queue - when work stealing occurs a thread processes removes a thread from victim processor’s queue. We refine this rough definition of work stealing implementation in Cilk-5 in the next subsection. We now give precise definitions of a multithreaded computation and related concepts.

A multithreaded computation is composed of a set of threads, each of which is a sequential ordering of unit-time instructions [2]. In a directed graph representation of the computation, instructions are connected by dependency edges, which provide a partial order of the execution. An execution schedule for P processors maps instructions to processors at a given computation step. Each execution schedule has a length - the number of steps it takes for the computation to complete with that schedule. During a computation, a thread (parent) may spawn other threads (children). The first instruction in the child depends on the instruction that spawned it in the parent thread - such edges in the graph are called spawn edges. Furthermore, a parent thread can have instructions that depend on an instruction in the child thread (sync is an example of this). Such edges are called join edges. For an execution schedule to exist, there can’t be a cyclic dependency in the instruction graph - Cilk-5 is even more restrictive than this by disallowing join edges from parents to children. A multithreaded computation is strict if all join edges in a thread point to an instruction in its ancestor. A multithreaded computation is fully strict if all join edges of a thread point to an instruction in its parent. It is easy to show that Cilk programs yield fully strict multithreaded computations (under the assumption that they don’t use concurrency primitives other than those defined by the language). We define work to be the total number of instructions in the directed acyclic graph of the multithreaded computation, and critical-path length as the length of the longest directed path in it. Note that the shortest possible execution schedule on the infinite number of processors is equal in length to the critical-path length.

Blumofe and Leiserson [2] show that when the work-stealing algorithm (as defined in their paper) is applied to a fully strict multithreaded computation, the total expected time of the execution on P processors is:

\[
T_P = T_1 / P + O(T_\infty)
\]

where \( T_1 \) is the total work and \( T_\infty \) is the critical path length. Their detailed analysis and proof are beyond the scope of this writeup. This result resembles Brent’s theorem [6], which says that if any multithreaded computation takes \( T_\infty \) time to complete on an infinite number of processors, then there exists a schedule for which the running time on \( P \) processors is

\[
T_P = T_1 / P + T_\infty \cdot (P - 1) / P^2.
\]

1 A fact that a schedule exists doesn’t mean there exists a scheduling algorithm that finds it efficiently.
From equation 1 we can define the critical-path overhead as the smallest constant \( c_\infty \) such that:

\[
T_P \leq T_1/P + c_\infty T_\infty
\]

(2)

In this inequality, we call the first term the work term and second the critical-path term. Now we observe that \( T_1 \) is the total number of instructions that have to be performed for a single processor to complete the multithreaded computation. However, \( T_1 \) is not the best time algorithm on a single processor, since it involves some overhead (\( \text{spawn} \) and \( \text{sync} \) have a cost greater than a simple method call) compared to the C elision which takes time \( T_S \). We define work overhead \( c_1 = T_1/T_S \).

Next, we claim that the critical path term is directly connected to work stealing. A greater critical path results in more work stealings, as shown in [2]. However, from experimental observations, in most applications critical path is small compared to total work, so work stealings are rare events. This roughly means that work stealings can be slightly more expensive, since they are infrequent. We define the average parallelism as \( \mathcal{P} = T_1/T_\infty \) - this is the maximum possible speedup of a program. Next, we define the parallel slackness as \( \mathcal{P}/P \). The assumption of parallel slackness is that \( \mathcal{P}/P \gg c_\infty \), that is, the number of processors is much smaller than the average parallelism, \emph{much smaller} defined with respect to \( c_\infty \). From the parallel slackness assumption it follows that \( T_1/P \gg c_\infty T_\infty \).

Substituting \( c_1 \) into 2 and using the parallel slackness assumption, we obtain:

\[
T_P \leq c_1 T_S/P + c_\infty T_\infty \approx c_1 T_S/T_P
\]

(3)

Equation 3 shows that \( T_P \) is mostly dependent on the work overhead - this drives the work first principle which says: \emph{Minimize the work overhead \( c_1 \), even at the expense of a larger critical-path overhead \( c_\infty \)}. In the next subsection we describe how this affects the design of the Cilk-5 compilation and scheduling.

C. Compiler and Work Stealing Implementation

A Cilk-5 compiler is run on a Cilk program to produce a regular C program which can be then compiled using a C compiler. It does so by producing two clones of each Cilk procedure - a fast clone and a slow clone. The fast clone is normally run when spawning a Cilk procedure. When a procedure gets stolen, the slow clone is called. We first describe the fast clone.

Fast clone for procedure \( P \) starts by allocating an activation frame which stores the pointer to the function, local variables values and the position label. For efficiency, a custom heap allocation is implemented. Before the fast clone normally returns, the frame is deallocated. If the fast clone \( P \) spawns a new procedure \( P_1 \), it stores the state of its variables to the activation frame and sets the position label to the point after the spawn. It then pushes the activation frame to the deque of that processor and does a normal C function call to the fast clone of the spawned procedure. After the call returns, it tries to pop the activation frame from the deque. If it succeeds, it stores the return value of the function and proceeds normally. If it doesn’t, that means that the activation frame for \( P \) has been stolen - since thefts occur on the opposite side of the deque than push operations, this implies all of the ancestor procedures have been stolen - \( P \) is now run by some other processor. In this case the procedure returns a dummy value, unwinds the call stack and returns control to the runtime. In the fast clone, all \( \text{sync} \) statements compile to no-ops - it can be easily observed that a fast clone never has any children while it is executing.

When a procedure gets stolen, it has previously been suspended and put on the deque. The slow clone first restores the values of the local variables, then reads the position label and jumps to the corresponding position with a goto. A \( \text{spawn} \) statement is translated just as in the fast clone. The \( \text{sync} \) statement is translated into a call to the runtime to check if there are any spawned children that haven’t yet returned.

Cilk-5 work stealing algorithm maintains the busy leaves property [2]. This means that a procedure with no spawned children is always assigned to some processor, and allows optimizations described above. Most of the time, \( \text{spawn} \) statements evaluate to few function calls.

The work stealing algorithm maintains a work stealing deque for each processor implemented as an array to which two pointers are maintained - tail \( T \) and head \( H \). A \( \text{push} \) operation amounts to writing the activation pointer to the location pointed by \( T \) and incrementing \( T \). A \( \text{pop} \) amounts to decrementing \( T \), and then checking if \( H > T \)- if it is, that means that a theft possibly occurred, so the worker acquires a lock \( L \) and repeats the protocol, failing to pop if \( H \) is still greater than \( T \). Thieves always acquire a lock \( L \) before incrementing \( H \) and checking if \( H > T \). Readers may wish to refer to [1] for pseudocode.

We can observe from this algorithm that, under the assumption of parallel slackness, work stealings occur rarely and most of the time processors run fast clones. Furthermore, pushes are only a few instructions and pops are fast most of the time. Steal operations involve acquiring a lock and are slow, but they occur rarely. This increases the critical-path overhead \( c_\infty \), but makes the work overhead \( c_1 \) as small as possible.

III. DECLARATIVE CONCURRENCY

Van Roy and Haridi describe how a declarative concurrency model [9] is implemented in the Mozart Programming System [10]. We describe their approach here, showing that it is possible to build high-level abstractions on top of simple declarative concepts.

We first define declarative concurrency. A concurrent program is declarative if all executions with a given set of inputs either all don’t terminate or they eventually reach partial
A single-assignment variable can be assigned only once after being declared. Assigning a value twice to a declarative variable raises a runtime error. It can be read any number of times. If it’s read before a value has been assigned, the computation stops until the variable is assigned a value. In a single-threaded computation this causes a deadlock, since there is no other thread to assign a value. Therefore, each thread observes a single value. This is useful for debugging - an observed undesired behaviour of the program can be repeated with the same set of inputs. There are many declarative models - we concentrate first on the data-driven (or supply-driven) concurrent model, which relies on the concepts of single-assignment variables and threads. Building on a model shown in table I similar to the simply-typed lambda calculus with extensions [14], we add single-assignment (or declarative) variables.

A single-assignment variable can be assigned only once after being declared. A variable may also be assigned to another variable, regardless of whether value has been already assigned to it or not. This is called variable-variable binding and creates a constraint that these variables must be equal. If the constraint cannot be fulfilled (e.g. due to variables already being assigned the same value), the binding results in a runtime error.

Next we add a linguistic construct for thread creation. An expression of the form $\text{thread} \ <a> \ end$ creates a new thread that evaluates expression $\langle a \rangle$ in parallel with the current one. We maintain a list of all threads in our abstract machine model. In each step, a scheduler picks a thread at random and that thread performs one computation step. Declarative variables are also called dataflow variables in the concurrent model.

Any concurrent program written in this model is declarative. Assume there is a program in our model that terminates (i.e. there are no runtime errors). If a thread $T$ in the program reads a dataflow variable, then in any execution schedule that variable will be either unbound in which case the thread waits until it becomes bound, or will already have a value. Since binding the variable by two different threads results in a runtime error and we excluded the possibility of runtime errors, the variable is bound at most one value. Since the program terminates, it is bound exactly one value (otherwise $T$ stalls forever). Therefore, each thread observes a single value for a variable it reads in all executions, so it always evaluates to the same value.

We now use this model to build higher-level programming abstractions. Most useful technique in the declarative concurrent model to communicate between threads are streams. A stream is a potentially unbounded list of messages. Streams are similar to functional lists in the sense that they are constructed using a `cons` operator which takes a head element and a tail list and returns a list. In dataflow programming, tail is not required to be bound to a value when a stream is declared. Threads that communicate through streams are called stream objects.

We show how stream objects can be used to implement an asynchronous producer/consumer pattern. One thread can use the stream to bind values to its tail, while the other can read them from the head. In the following example, one thread creates a stream of integers and the other calculates their sum:

$$\text{fun (Generate N Limit)}$$
$$\text{if (N < Limit) then N} \ | \ \{\text{Generate N+1 Limit}\}$$
$$\text{else nil end}$$

$$\text{fun (Sum Xs A)}$$
$$\text{case Xs of X | Xr then \{Sum Xs A+X\}}$$
$$\text{else A end}$$

$$\text{local Xs S in}$$
$$\text{thread Xs = \{Generate 0 150000\} end}$$
$$\text{thread S = \{Sum Xs 0\} end}$$

The producer thread starts by creating a new stream with an unbound tail and binds it to the output variable $Xs$. It then calls `Generate` recursively. The consumer reads a dataflow variable $Xs$ and tries to pattern match it against a non-empty stream. Pattern matching counts as reading a variable, so the consumer stalls until $Xs$ becomes bound. When it does and if a stream is non-empty, the consumer sums the head value and calls `Sum` recursively on the tail of the stream. The consumer will always wait until the new values from the producer become available. It’s also possible to create another stream object that reads the producer stream and produces a new one for the consumer (e.g. it could filter the elements). Such a stream object is called a transducer.

A stream can be read by multiple threads, since they don’t actually consume the stream. On the other hand, only one thread can write to the stream, since writing means assigning to the unbound tail. Allowing multiple threads to write to the stream results in binding the same variable twice - a runtime error. In fact, allowing this would make it possible to write non-declarative programs.

A stream can be handled using a higher-order iterator. Function `Fold` is an example - it takes a stream, an initial value and a function:

$$\text{fun (Fold Xs Init F)}$$
$$\text{case Xs of X | Xr}$$

3 We treat this as non-termination.
4 Depending on the language, this is either detected at compile-time, results in a runtime error or undefined behaviour.

5 In Oz procedures can have output values which are annotated with a prefix `. Functions annotation `fun` is syntactic sugar for procedures with only a single output value.
EDIC RESEARCH PROPOSAL 5

It reads a value from the stream, applies the specified function to initial value and the stream value and then calls itself recursively with the tail of the stream, the same function and the computed value as the initial value. If the stream is empty, it just returns the initial value. For instance, in the previous example, the \texttt{Sum} function can be replaced with a call to \texttt{Fold} where the initial value is 0 and the function is integer addition. Other higher-order stream methods can be implemented in a similar way.

One problem with the producer/consumer pattern as described above is that if the producer creates elements quicker than the consumer can process them, the elements will pile up and consume available memory. We only want the producer to create additional elements if there are less than a certain number of them. This is done using a special transducer called a \textit{bounded buffer}. This warrants modifications in the producer, since the producer may create new elements only if the bounded buffer allows it - checking whether the buffer can handle more elements is done by reading a stream datalflow variable. Once the stream variable is bound by the buffer, the producer can bind a newly created element to the head of the stream, and recursively repeat this on the tail. For brevity we omit the code here and refer readers to [9].

Dataflow variables can also be used to implement barriers. For instance, for\texttt{fork/join} style of computation [1] [5] we discussed in the section on Cilk-5 language is a subset of this model. Each task spawned by the \texttt{thread} statement can bind a dataflow variable once its done. The parent task can implement \texttt{sync} by reading all of these variables sequentially.

A stream object is implemented with a thread running a recursive procedure \texttt{Next} which communicates with other stream objects using streams:

\begin{verbatim}
  proc {StreamObject In State ?Out}
    case In of InH | InT then OutH OutT NState in
    (Next State InH NState OutH) Out = OutH | OutT
    (StreamObject InT NState OutT) [] nil then Out = nil end
  end
  thread {StreamObject Input Start Output} end
\end{verbatim}

In the example above we assume that undefined variables and procedures have been defined in elsewhere scope. A stream object maintains an inner state and calculates the next state and the output value based on the input value and the current state. In a high-level language this and other abstractions we’ve discussed can be provided as separate linguistic constructs.

The data-driven model presented so far can be extended by introducing lazy execution, yielding a \textit{demand-driven} model. This is done by adding the statement \texttt{(ByNeed P X)} which binds the variable \texttt{X} to the result of the procedure \texttt{P} called in a separate thread, but does so only if some thread reads \texttt{X}. This extension can be used to define lazy streams which evaluate their tail only when some thread reads it, as well as lazy functions on streams [9].

Declarative concurrency has certain limitations. First of all, some applications require nondeterminism which cannot be expressed in this model. For instance, it’s not possible to have multiple threads that read the same stream and each consumes a separate object from the stream, as this is the source of potential nondeterminism - all threads have to read all the elements of the stream. Another example is the client/server pattern. It’s not possible for two clients to send requests to the server simultaneously. They cannot use the same stream, since that would mean binding the tail twice. They could use two separate streams, but the server then wouldn’t know which one to read - reading either of them or reading them in some order can stall until a value appears, thus potentially starving the other client.

Nondeterminism can be introduced with a \texttt{(IsDet X)} statement, which checks whether a variable is already bound, returning a boolean. This statement potentially leads to two different computations depending on whether the variable has already been bound or not. If another thread is supposed to bind the variable, depending on the execution schedule this will yield different computations.

Another issue with declarative concurrency are runtime errors due to binding the same variable twice. What’s important for declarative programs is that if the program fails for some execution schedule, then it should fail for all of them. Although we might not observe the same traces on subsequent program runs, this is indeed the case in the model we’ve presented. On the other hand, sometimes we might not want to crash an entire application due to a bug local to some component, so we can introduce \texttt{try-catch} style exception handling. Imagine a try-catch block around the assignment to the same dataflow variable in two different threads. Depending on the execution schedule, different values are assigned to the variable - determinism is lost.

The benefits of this model is that it provides powerful primitives for deterministic concurrent programming which can be used for communication and synchronization, simplifying existing techniques and allowing a plethora of new ones.

IV. POLYMORPHIC EMBEDDINGS

Carette, Kiselyov and Shan [7] analyze the problem of embedding a domain-specific language (\textit{object language}) into a host language (\textit{metalanguage}). In the classical approach, terms in the object language are commonly represented using objects in the host language (\textit{tags}). Carette, Kiselyov and Shan show how a simply typed lambda calculus with extensions [14] can be embedded into a language with a Hindley-Milner type system, such as OCaml. They rule out meaningless terms by using the typechecker of the host language - this ensures interpreters are faster and don’t get stuck.

In OCaml, terms of the object language can be represented by algebraic data types. Terms constructed this way can be evaluated by an interpreter function \texttt{eval}:

\begin{verbatim}
  type var = VZ | VS of var;;
  type exp = V of var | B of bool |
            L of exp * A of exp * exp;;
  let rec lookup(x::env) =
    function VZ -> x | VS v -> lookup env v;;
\end{verbatim}

We can represent a lambda calculus term \((\lambda x.x)true\) with A (L (V VZ), B true), where we use unary de Bruijn indices to represent variables. The lookup function finds the appropriate variable in the environment. The universal type \(u\) used because eval function would otherwise return results of different types depending on the match. Function eval takes the environment, matches the terms and interprets it.

This approach has several problems. First of all, there is no way to check if the variable corresponding to the provided de Bruijn index exists in the environment - the interpreter may fail to pattern match. Second, the match in the eval function is not exhaustive. A term A (B true, B false) fails at runtime since it doesn’t provide a function for the application. Another problem is that such an interpreter is slower because it has to pattern match against tags. To solve these problems, Carette, Kiselyov and Shan propose representing object programs using ordinary functions instead of data constructors:

```ocaml
let varZ env = fst env;;
let varS vp env = vp (snd env);;
let b (bv: bool) env = bv;;
let lam e env = \x -> e (x, env);;
let app e1 e2 env = (e1 env) (e2 env);;

The benefit of this is that the type system of the host language can rule out meaningless terms. We can now represent term \(\lambda x.x\)true as

```

This expression is actually a partially applied function in OCaml and can be evaluated by applying it to an empty environment: test1 = app (lam varZ) (b true). This term such as app (b false) (b true) is rejected by the type checker.

The presented interpreter successfully embeds an object language into a metalanguage. However, we want to be able to process the same term in various ways. Carette, Kiselyov and Shan show how interpreter behaviour can be abstracted, and implement a compiler and a partial evaluator.

To abstract the object language syntax from its semantics, we use OCaml module signature construct. The module signature Symantics defines a parametrized abstract type member repr which serves as a representation type for different terms and abstract functions for various object language terms. Here we show only some of those presented in the original paper:

```ocaml
module C = struct
  type ('c, t) code
  val int : unit -> 'x
  val bool : bool -> 'x
  val lam : ('c, 'dv) C.rep -> ('c, 'dv) R.rep
  val add : ('c, int) rep -> ('c, int) rep
  val if_ : ('c, bool) rep ->

module type Symantics = sig
  type ('c, 'dv) rep
  val int : int -> ('c, int) rep
  val bool : bool -> ('c, bool) rep
  val lam : ('c, 'da) rep -> ('c, 'db) rep
  val app : ('c, 'da -> 'db) rep -> ('c, 'db) rep
  val add : ('c, int) rep -> ('c, int) rep
  val if_ : ('c, bool) rep ->

end
```

It’s worth mentioning that since OCaml is a call-by-value language, the two cases in the \(if\_) term are passed as thunk types. The type parameter \(c\) for the abstract type \(\text{rep}\) is an environment classifier that will be needed for MetaOCaml code generation later, while \(dv\) is the type of the staged expression (e.g., a staged boolean expression has type \(\langle 'c, bool \rangle\)). An interpreter that evaluates an object term to its value in the metalanguage defines the \(\text{rep}\) type to be the underlying type. We show a partial implementation here:

```ocaml
module R = struct
  type ('c, 'dv) rep = 'dv
  let bool (b: bool) = b
  let lam f = f
  let app e1 e2 = e1 e2
end
```

Module signature Symantics can be implemented as a compiler which produces code in a target language. The authors use MetaOCaml [8] staged expressions as a target language. MetaOCaml represents a staged expression \(e\) of type \(t\) as a value of type \(\langle 'c, t \rangle\) code, which is created by the bracketnotation \("<e>....\)\). Escape notation \." within a bracket allows smaller staged expressions to be used together. This doesn’t actually evaluate a staged expression, but allows using it as if it were an ordinary value. To compile and run the staged expression, the \(run\) notation \."\(e\) is used. We again show a partial implementation:

```ocaml
module C = struct
  type ('c, 'dv) code
  val int : unit -> 'x
  val bool : bool -> 'x
  val lam : ('c, 'dv) C.rep -> ('c, 'dv) R.rep
  val add : ('c, int) rep -> ('c, int) rep
  val if_ : ('c, bool) rep ->

end
```

The authors then show how to write a partial evaluator using the ideas represented above. The partial evaluator relies on the implementations of the interpreter \(R\) and the compiler \(C\). The partial evaluator should carry as much information as possible about the terms, so it’s representation type should be either a statically known value or a dynamic staged term. The first attempt is to define the representation type using data-constructors like this:

```ocaml
module type Symantics = sig
  type ('c, 'dv) rep
  val int : int -> ('c, int) rep
  val bool : bool -> ('c, bool) rep
  val lam : ('c, 'da) rep -> ('c, 'db) rep
  val app : ('c, 'da -> 'db) rep -> ('c, 'db) rep
  val add : ('c, int) rep -> ('c, int) rep
  val if_ : ('c, bool) rep ->

end
```

Using this definition of \(\text{rep}\) integer and boolean literals yield statically known values of \(\text{S0}\). Addition \(\text{add}\) does a pattern match to add statically known terms, and produces a dynamic term otherwise (unlike before, this pattern match is exhaustive). However, in case \(\text{if}\_)\) has a dynamic term for the condition, and a static term for one of the branches, the evaluator should produce a dynamic term. Evaluator should lift the static term of type \(\langle 'c, 'a \rangle\) \(\text{R.rep}\) to a dynamic term of type \(\langle 'c, 'a \rangle\) \(\text{C.rep}\). However, the module Symantics only defines \(\text{iff}\) functions for integers and booleans, and not arbitrary types.

The only way to create the dynamic term for an arbitrary type is to build it gradually, so the next attempt is to build it...
for both static and dynamic terms - even if the static term is known, the evaluator still builds the dynamic term. We also define a function \( \text{absstr} \) which extracts the dynamic term:

\[
\text{type} \quad ((\text{c}, \text{dv}) \text{ rep} = \\
\P1 \text{of} \quad ((\text{c}, \text{dv}) \text{ R.rep option}} \quad \ast \ast \text{ (c', dv) C.rep} \\
\text{let} \quad \text{absstr} \text{ (P1 (_, dyn))} = \text{ dyn}
\]

With this definition the dynamic type of each term is always constructed and the problem with \( \text{if}_n \) is solved. Lets observe functions - we can pass a statically known function to a dynamically known one as in \( \lambda f.\text{f}(\lambda x.n) \), obtaining only a dynamic term. We can also simplify terms in which a statically known function is applied to a static argument, as in \((\lambda x.n)\text{true} \). However, we cannot simplify terms such as \( \lambda n.\text{true} \) to \( \lambda n.n \), since \( n \) is not statically known - \( \lambda x.n \) has type \((\text{c', da} \rightarrow \text{db}) \text{ rep} \), that is \( \P1 \text{ (Some (da} \rightarrow \text{db}, (\text{c', da} \rightarrow \text{db}) \text{ code} \) \}. If the static type were \((\text{c, da}) \text{ rep} \rightarrow (\text{c', db}) \text{ rep} \) it would be possible to apply it to \( n \). This is solved by modifying the module signature Symantics to have the abstract type \( \text{rep} \) take the third type parameter \( \text{sv} \) used to store the static type of terms. We define \( \text{lam} \) in Symantics as:

\[
\text{val lam: ((c', sa, da) rep} \rightarrow \\
\text{(c', sb, db) rep as } \text{x}) \rightarrow \\
\text{(c', x, da } \rightarrow \text{db) rep}
\]

This ensures that the enough type information is preserved so that the static function term can be applied to statically unknown terms. For the complete definition of the partial evaluator we refer readers to [7].

V. DISCUSSION AND THESIS PLAN

We presented a concurrent programming language Cilk-5 with an efficient scheduler tailored for that specific domain [1]. Another concurrent model we’ve presented is more powerful and relies on the concept of dataflow variables to preserve deterministic behaviour [9]. We’ve also shown how its possible to embed an object language within a host language [7]. Although papers presented here offer solutions to many problems, they leave a number of open questions.

Leiserson and Blumofe [2] give spatial and time complexity guarantees for their scheduling algorithm only if its used for fully strict multithreaded computations. Dataflow programming is more general and doesn’t always yield even strict multithreaded computations. It is not obvious whether work stealing can be used in the dataflow setting with good time complexity.

We plan to implement a programming language with a deterministic programming model based on dataflow variables and enrich it with various high-level constructs, while preserving determinism. One possibly useful extension are multiple assignment variables. Although variables are known to cause data races in a concurrent setting, we can encode them in the declarative concurrency model we’ve presented:

\[
\text{procedure Loop \{In \ Out\} \quad \text{case In of X\|Int \ then \ begin \}
\]

The procedure above waits for the stream to get bound, then reads the head of the input stream and writes the value to the output stream. Such a thread can be used to simulate a variable. This technique is inspired by [15]. There are 2 restrictions that follow from our model. First, the input stream can be written by only one thread. This means that only one thread can access the variable by reading the output stream. Second, we are required to read from the tail of the output stream each time we write to the input stream. The latter inconvenience can be overcome by introducing multiple assignment variables. To preserve determinism, we should only ensure that other threads can never access a variable local to some other thread. This allows using classical imperative constructs such as while-loops.

Next, we can use streams of pairs to implement dataflow maps. Once we are done binding key/value pairs to the tail of the stream, we can bind \text{nil} to it, disallowing addition of new elements. Other threads can wait for a specific key to appear in the stream by pattern matching the tail of the stream recursively until the key appears or the stream is sealed. We can provide syntactic sugar to use this pattern - a map to which keys can be added, but not removed and which can be sealed at some point. In fact, we can obtain a more powerful model by allowing multiple threads to add key/value pairs - this cannot be encoded in the previous model, but the determinism persists if there is no seal operation. This is somewhat similar to ideas proposed by Gelernter [12], with the difference that tuples cannot be removed.

Following the ideas in [11] [13], we also plan to add support for dataflow arrays and initialization comprehensions. Pattern matching constructs for arrays and dataflow maps is another relevant concept which should be defined in a way that preserves determinism, as it may not hold in general (e.g. pattern matching on a value in the dataflow map regardless of the key value can yield nondeterministic computations).

We plan to first implement the dataflow language presented here as a library which relies on efficient scheduling algorithms. Building on ideas presented in [7], we plan to provide a high-level embedding of our dataflow language using Scala as a host language. This will allow using the staged representation of programs written in this language to execute them on multiple target parallel platforms, such as multicore processors architectures, supercomputers and computer clusters.

REFERENCES


Solving Large, Irregular Graph Problems Using Adaptive Work-Stealing, 
Proceedings of the 2008 37th International Conference on Parallel Pro-
cessing, 2008.

Y. Zhou, Cilk: An Efficient Multithreaded Runtime System, SIGPLAN 
Not., 1995.


Tagless Staged Interpreters for Simpler Typed Languages, Journal of 

[8] W. Taha, A Gentle Introduction to Multi-stage Programming, Domain-
specific Program Generation, LNCS, 2004.


[10] P.V. Roy, S. Haridi, Mozart: A Programming System for Agent Applica-
tions, International Workshop on Distributed and Internet Programming 
with Logic and Constraint Languages, 1999.

Parallel Computing, ACM Transactions on Programming Languages and 

on Programming Languages and Systems, 1985.

Sisal Model of functional programming and its implementation, Second 
Aizu International Symposium on Parallel Algorithms/Architectures Syn-


[15] M. Odersky, Functional Nets, European Symposium on Programming, 
2000.