Future Contracts *

Christos Dimoulas Riccardo Pucella Matthias Felleisen
Northeastern University, Boston, MA

Abstract

Many recent research projects focus on language support for behavioral software contracts, that is, assertions that govern the boundaries between software building blocks such as procedures, classes, or modules. Contracts primarily help locate bugs in programs, but they also tend to affect the performance of the program, especially as they become complex.

In this paper, we introduce `future` contracts and parallel contract checking: software contracts annotated with `future` are checked in parallel with the main program, exploiting the now-common multiple-core architecture. We present both a model and a prototype implementation of our language design. Our model comprises a higher-order imperative language and we use it to prove the correctness of our design. Our implementation is robust enough to measure the performance of reasonably large benchmarks, demonstrating that the use of `future` contracts can lead to significant performance improvements.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]

General Terms Design; Reliability

Keywords contracts, higher-order functions, behavioral specifications, software reliability

1. Introduction

Programmers frequently insert assertions into their programs to ensure that at various points during the execution, the program state satisfies some important logical property. For multi-component systems, assertions have become a popular tool to express constraints on the interface between two components that a static type system cannot express. When used in this context assertions are called *behavioral software contracts* or just contracts. While Parnas (1972) introduced contracts as early as 1972, they only became popular with Eiffel (Meyer 1988, 1992). Later, contracts were introduced on many other platforms (Duncan and Hoelzle 1998; Gomes et al. 1996; Holt et al. 1987; Karaorman et al. 1999; Kölling and Rosenberg 1997; Kramer 1998; Luckham and Henke 1985; Ploesch and Pichler 1999; Rosenblum 1995). Most recently, Findler and Felleisen (2002) have introduced contracts into the world of higher-order functional programming languages.

Naively speaking, contracts monitor the flow of values across component boundaries. Contracts remain invisible as long as the constraints they express are satisfied. When the constraints are not satisfied, the contract monitoring system signals a contract violation, pinpointing the violation's origin, explaining how the contract is violated, and most importantly assigning blame to the violating component. Blame assignment helps programmers isolate errors and gets them started with their debugging efforts.

Unfortunately, monitoring contracts tends to impose a significant run-time overhead on the program's execution, especially for defensive programmers wanting to write precise contracts. In the worst case, contract monitoring can affect the algorithmic complexity of a function even for careful programmers, although Findler et al. (2007) have recently demonstrated how to overcome this problem in many cases. In the average case, contract monitoring consumes a substantial, though constant amount of time and space. As a result, many programmers turn off contract monitoring or relax contracts so that they monitor fewer properties than desirable.

We have gathered anecdotal evidence for this last observation. A survey of PLT Scheme users about whether they wrote applications with complex contracts confirmed that programmers do not generally use the contracts they would like, but rather settle for much lighter contracts. In fact, contracts are often used only in the debugging phase of software development and then, to avoid contract-checking overhead, they are removed from production code, exactly where contracts are at their most useful. Thus, programmers' fear of the run-time cost of contract checking leads to less reliable software.

Multi-core architectures suggest an obvious way to address this problem, namely by monitoring contracts in parallel with program execution. Since software contracts are usually functional computations—even in imperative languages—running them in parallel should be easy and should save some of the cost of evaluating them. Because communication between threads is not free, evaluating every single contract in parallel is not cost effective. Indeed simple experiments validate this intuition and show that evaluating every contract in parallel lead to a slow down in execution time. This impact is worsened by the presence of effects in the main program and in contracts, which require synchronization between the main program thread and the contract monitoring thread.

The question, then, is how to choose the contracts that benefit from parallel evaluation. To help answer this question, we introduce the notion of *future contracts* inspired by Halstead's (1984) future construct. Following Halstead's work, the annotation `future` on a contract indicates that a contract should be checked in parallel with the rest of the program; unannotated contracts are executed in-line. While Halstead's future expressions immediately produce a

* This work was supported in part by AFOSR grant FA9550-09-1-0110. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the AFOSR or the U.S. Government.

PPDP'09, September 7–9, 2009, Coimbra, Portugal.

future value, a `future` contract produces nothing. Instead, the main thread sends such a contract to the monitoring thread and proceeds until it must perform an observable operation (e.g., a side-effect or an I/O operation). At that point, the main thread waits to ensure that the contract succeeds. If so, the computation proceeds; otherwise the program terminates with a contract exception.

The introduction of `future` contracts into a mostly functional language poses both theoretical and practical challenges. Given the critical nature of software contracts in the development process, we naturally wish to ensure that parallel contract execution does not undermine contract monitoring. This is especially important here since we are potentially adding concurrency to the language, and ensuring correctness of concurrent programs is well-known to be subtle. We therefore develop a model of parallel contract execution for a higher-order and imperative programming language and prove that it is semantically equivalent to sequential contract monitoring, including blame assignment.

Of course, a theoretical result is interesting only when it is backed up with an implementation that confirms the practical part of the conjecture. We have implemented our model in the context of PLT Scheme's contract system. The core of our implementation is a carefully crafted communication infrastructure, ensuring that our prototype is sufficiently efficient to conduct a number of performance tests. These tests confirm the usefulness of `future` contracts, and help us formulate some relatively straightforward guidelines concerning the use of `future` contracts that we hope will encourage the wide use of comprehensive contracts in software development.

In the next section we introduce a model of parallel contract checking in a higher-order imperative and parallel world. Then we proceed to present the correctness theorem, the design of the communication architecture, the prototype implementation, and a set of benchmarks that validate our conjectures concerning the performance improvements of contract monitoring.

2. λ^{con}: A Functional Language with Contracts

The language of λ^{con} is an A-normal form λ-calculus. The language includes observable side effects through the use of an output operator. We generalize effects to store-manipulation effects in section 4, but most subtleties already arise for simple output.

Figure 1 presents the syntax of λ^{con}. A program D `in` e consists of a sequence of recursive function definitions D and a main expression e. Intuitively, the functions in D are the functions that can be used in e, and those functions are protected (from each other) by contracts on the functions' arguments and return values.

The language is typed, but to simplify the presentation in this paper, we do not present the type system, which is completely standard. We simply assume that we are given only programs that are well-typed with respect to a standard type system.

Contracts guard the flow of values. Flat contracts of the form $\texttt{contract}(v)$, where v is a Boolean-valued function, guard first-order values. A value u satisfies the contract if the contract computation $(v\ u)$ yields `true`. Higher-order contracts of the form $c_1 \mapsto c_2$ guard functions; the domain contract c_1 enforces that the argument to the function satisfies specified properties, while the range contract c_2 enforces constraints for the result of the function.

The following examples provide a taste of contracts:

$$sqrt : \texttt{contract}(\lambda x.x \geq \ulcorner 0 \urcorner) \mapsto \texttt{contract}(\lambda x.x \geq \ulcorner 0 \urcorner)$$

where $\ulcorner i \urcorner$ is the integer literal representation of integer i in λ^{con}. That is, $sqrt$ is a function that computes the square root of non-negative numbers. Its contract guarantees that its inputs and results are at least 0. Here is a second example:

$$encode : (\texttt{contract}(prime?) \mapsto \texttt{contract}(prime?)) \\ \mapsto (\texttt{contract}(string?) \mapsto \texttt{contract}(string?))$$

That is, $encode$ is a higher-order function that expects a function from prime numbers to prime numbers, and that returns another function from strings to strings.

The key innovation of λ^{con} is the introduction of the contract combinator $\texttt{future}(c)$, which indicates that the contract c should be checked in parallel with the main program. Semantically, $\texttt{future}(c)$ is taken to be equivalent to c.

Contracts are enforced at run time. To record that a value is guarded by a contract during execution, we use *obligation* expressions of the form $w^{c,pos,neg}$. Superscripts pos and neg are critical for contracts guarding functions; they enable the contract checking monitor to properly assign blame in the case of contract failure. The first superscript—the positive blame position—captures the party responsible for the values produced by a function, while the second superscript—the negative blame position—captures the party responsible for the values provided to the function. The responsible party for a contract failure can be identified by carefully updating these superscripts as an obligation flows through a program.

Expressions in λ^{con} are standard for an A-normal form λ-calculus, with the addition of two new expression forms. Expression $\texttt{output}(v); e$ outputs value v before continuing as e. Expression $\texttt{check}(e_1); e_2$ is an explicit contract checking expression: e_1 is an expression representing a contract that must be checked before expression e_2 evaluates. The result value of e_1 is discarded—if the contract does not evaluate to `true`, then it reports an error that terminates the program. This explicit contract checking construct enables the parallel checking of the contract; in this section, however, it serves only as a sequencing operator.

The function identifiers declared in D may be used in the main expression e or functions in D. To force the contracts associated with each such function identifier to be checked at runtime, we compile the program by replacing every function identifier f appearing in e or other parts in D by an obligation where the reference to f is guarded by the contract associated with f in D. This compilation process allows us to appropriately tag each obligation with blame labels (Findler and Felleisen 2002). The positive blame position of the obligation is simply the name f of the function, while the negative blame position is the name of the caller.

Our goal now is to design the semantics of our language so that we can describe precisely the effectful behavior of programs. We are especially interested in nonterminating programs that produce infinitely many outputs, such as infinite loops with output statements in the body of the loop.

Anticipating the development in the following sections, we define a two-level reduction relation representing the execution of programs: a local reduction relation $\rightarrow$ regulating the evaluation of pure expressions, and a global reduction relation $\Rightarrow$ regulating the evaluation of effectful expressions. This is similar to the semantics of CML (Reppy 1999) and other concurrent languages.

The local reduction relation $D \vdash e \longrightarrow e'$ (or $D \vdash e \longrightarrow \texttt{error}(str)$) regulates the evaluation of simple expressions not involving output or explicit contract checking. The rules appear in figure 2. An obligation with a flat contract reduces to an `if` expression that checks if the guarded value satisfies the contract. The contract check is performed first. If the contract check fails, the execution terminates with an error message blaming the identifier in positive position. If the contract check succeeds, execution proceeds by returning the previously guarded value to its context without a contract. An application of a value to a function with a higher-order contract reduces to a check of the domain contract on the argument

Programs	p	::=	D `in` e
Declarations	D	::=	$d \ldots d$
	d	::=	`val rec` $x : c = v$
Atomic Expressions	w	::=	$x \mid v$
Expressions	e	::=	$w \mid (w\ \ w) \mid$ `let` $x = e$ `in` $e \mid w\ aop\ w \mid w\ rop\ w \mid$ `string?`$(w) \mid w{::}w \mid$ `hd`$(w) \mid$ `tl`$(w) \mid$ `mt`$(w) \mid$ `if` $w\ e\ e \mid$ `check`$(e); e \mid$ `output`$(w); e \mid$ $w^{c,str,str} \mid$ `blame`(str)
Relational Operators	rop	::=	$= \mid \geq$
Arithmetic Operators	aop	::=	$+ \mid * \mid - \mid /$
Variables	x	$\in$	*Variables*
Constants	j	$\in$	*Integers*
	str	$\in$	*Strings*
Evaluation Contexts	F	::=	$[\,] \mid$ `let` $x = F$ `in` e
	E	::=	$F \mid$ `let` $x = E$ `in` $e \mid$ `check`$(E); e$
Values	v	::=	$\lambda x.e \mid \ulcorner i \urcorner \mid str \mid$ `true` $\mid$ `false` $\mid$ `nil` $\mid v{::}v \mid v^{c \mapsto c,str,str}$
Contract Values	c	::=	`contract`$(v) \mid c \mapsto c \mid$ `future`(c)

Figure 1. Syntax of λ^{con}

$D \vdash$	$\ulcorner j_1 \urcorner + \ulcorner j_2 \urcorner$	$\longrightarrow$	$\ulcorner j_1 + j_2 \urcorner$	
$D \vdash$	$\ulcorner j_1 \urcorner * \ulcorner j_2 \urcorner$	$\longrightarrow$	$\ulcorner j_1 * j_2 \urcorner$	
$D \vdash$	$\ulcorner j_1 \urcorner - \ulcorner j_2 \urcorner$	$\longrightarrow$	$\ulcorner j_1 - j_2 \urcorner$	
$D \vdash$	$\ulcorner j_1 \urcorner / \ulcorner 0 \urcorner$	$\longrightarrow$	`error`$(\text{"}/0\text{"})$	
$D \vdash$	$\ulcorner j_1 \urcorner / \ulcorner j_2 \urcorner$	$\longrightarrow$	$\ulcorner j_1/j_2 \urcorner$	$j_1 \neq 0$
$D \vdash$	$\ulcorner j_1 \urcorner \geq \ulcorner j_2 \urcorner$	$\longrightarrow$	`true`	if $j_1 \geq j_2$
$D \vdash$	$\ulcorner j_1 \urcorner \geq \ulcorner j_2 \urcorner$	$\longrightarrow$	`false`	if $j_1 < j_2$
$D \vdash$	$\ulcorner j_1 \urcorner = \ulcorner j_2 \urcorner$	$\longrightarrow$	`true`	if $j_1 = j_2$
$D \vdash$	$\ulcorner j_1 \urcorner = \ulcorner j_2 \urcorner$	$\longrightarrow$	`false`	if $j_1 \neq j_2$
$D \vdash$	`string?`(v)	$\longrightarrow$	`true`	if $v \in String$
$D \vdash$	`string?`(v)	$\longrightarrow$	`false`	if $v \notin String$
$D \vdash$	$(\lambda x.e\ \ v)$	$\longrightarrow$	$e[x/v]$	
$D \vdash$	`let` $x = v$ `in` e	$\longrightarrow$	$e[x/v]$	
$D \vdash$	x	$\longrightarrow$	v	where $(\texttt{val rec}\ x : c = v) \in D$
$D \vdash$	`if true` $e_1\ \ e_2$	$\longrightarrow$	e_1	
$D \vdash$	`if false` $e_1\ \ e_2$	$\longrightarrow$	e_2	
$D \vdash$	`mt`$(\texttt{nil})$	$\longrightarrow$	`true`	
$D \vdash$	`mt`$(v_1{::}v_2)$	$\longrightarrow$	`false`	
$D \vdash$	`hd`$(\texttt{nil})$	$\longrightarrow$	`error`$(\text{"}hd\text{"})$	
$D \vdash$	`hd`$(v_1{::}v_2)$	$\longrightarrow$	v_1	
$D \vdash$	`tl`$(\texttt{nil})$	$\longrightarrow$	`error`$(\text{"}tl\text{"})$	
$D \vdash$	`tl`$(v_1{::}v_2)$	$\longrightarrow$	v_2	
$D \vdash$	`blame`(str)	$\longrightarrow$	`error`(str)	
$D \vdash$	$v_1^{\texttt{contract}(v_2),pos,neg}$	$\longrightarrow$	`let` $x = (v_2\ \ v_1)$ `in if` $x\ v_1$ `blame`(pos)	
$D \vdash$	$(v_1^{c_1 \mapsto c_2,pos,neg}\ \ v_2)$	$\longrightarrow$	`let` $x = v_2^{c_1,neg,pos}$ `in let` $y = (v_1\ \ x)$ `in` $y^{c_2,pos,neg}$	
$D \vdash$	$v_1^{\texttt{future}(\texttt{contract}(v_2)),pos,neg}$	$\longrightarrow$	`check`$(\texttt{let}\ x = (v_2\ \ v_1)\ \texttt{in if}\ x\ v_1\ \texttt{blame}(pos)); v_1$	
$D \vdash$	$(v_1^{\texttt{future}(c_1 \mapsto c_2),pos,neg}\ \ v_2)$	$\longrightarrow$	`let` $x = v_2^{\texttt{future}(c_1),neg,pos}$ `in let` $y = (v_1\ \ x)$ `in` $y^{\texttt{future}(c_2),pos,neg}$	

Figure 2. Reduction rules for local evaluation

value and a check of the range contract on the result of the application. The positive and negative blame identifiers are reversed for the argument check ensuring that contravariant portions of the contract are blamed correctly. An obligation with `future`(c) where c is a flat contract reduces to a check of c on the guarded value using the `check`$(e_1); e_2$ operator. If c is a higher-order contract, checking is postponed as in the $c_1 \mapsto c_2$ case; the two newly introduced contracts inherit the `future` annotation. In later sections, when we give `check`$(e_1); e_2$ a concurrent interpretation, `future`(c) becomes a parallel contract checking annotation.

The global reduction relation regulates the evaluation of expressions involving outputs and explicit contract checking, and is defined over *states*. A state $e \parallel \varphi$ is a combination of an expression e and a output trace φ. To avoid having to reason about infinite sequences of outputs, we follow the standard approach of approximating an infinite sequence by its finite prefixes. It is technically convenient to define a family of context-sensitive small-step reduction relations (Felleisen and Hieb 1992) parameterized by a natural number i standing for the maximum number of outputs allowed in the computation. Intuitively each one of those reduction relations describes the behavior of programs up to a finite trace output of size i. These global reduction relations take the form $D \vdash \sigma \overset{i}{\Longrightarrow} \sigma'$, stating that state σ reduces to σ' in the context of declarations D, and that the output traces that appear in σ and σ' are smaller than i. The rules for this relation appear in Figure 3. The checking reduction discards the value v and the execution proceeds with

$$
\begin{array}{llllll}
D & \vdash & E[e] \parallel \varphi & \stackrel{i}{\Longrightarrow} & E[e'] \parallel \varphi & \text{where } D \vdash e \longrightarrow e' \\
D & \vdash & E[\texttt{check}(v); e] \parallel \varphi & \stackrel{i}{\Longrightarrow} & E[e] \parallel \varphi & \\
D & \vdash & E[\texttt{output}(v); e] \parallel \varphi & \stackrel{i}{\Longrightarrow} & E[e] \parallel \varphi; v & \text{if } |\varphi| < i \\
D & \vdash & E[\texttt{output}(v); e] \parallel \varphi & \stackrel{i}{\Longrightarrow} & \top \parallel \varphi & \text{if } |\varphi| = i \\
D & \vdash & E[e] \parallel \varphi & \stackrel{i}{\Longrightarrow} & \texttt{error}(str) \parallel \varphi & \text{where } D \vdash e \longrightarrow \texttt{error}(str)
\end{array}
$$

Figure 3. Reduction rules for computation propagation, `output`, and `check`

$$
eval^i(D \texttt{ in } e) = \begin{cases} \sigma_f & \text{if } D \vdash e \parallel \varnothing \stackrel{i}{\Longrightarrow}{}^* \sigma_f \\ \bot \parallel \varphi & \text{if } \forall e' \text{ such that } D \vdash e \parallel \varnothing \stackrel{i}{\Longrightarrow}{}^* e' \parallel \varphi,\ \exists e'' \text{ such that } D \vdash e' \parallel \varphi \stackrel{i}{\Longrightarrow} e'' \parallel \varphi \end{cases}
$$

$$
eval(D \texttt{ in } e) = \begin{cases} r & \text{if } \exists i \in \mathbb{N} \text{ such that } eval^i(D \texttt{ in } e) = r \text{ and } r \neq \top \parallel \varphi \\ \bot \parallel \hat{\varphi} & \text{if } \forall i \in \mathbb{N}\ eval^i(D \texttt{ in } e) = \top \parallel \varphi_i \text{ and } \hat{\varphi} \text{ is the smallest (possibly infinite) trace} \\ & \text{such that } \varphi_0 \sqsubseteq \varphi_1 \sqsubseteq \varphi_2 \sqsubseteq \cdots \sqsubseteq \hat{\varphi} \end{cases}
$$

Figure 4. λ^{con} evaluators

the evaluation of e_2. The output reduction additionally appends the value v to the trace φ of the resulting state. If the size of the trace φ of the state before the reduction step is already equal to i then the evaluation terminates with $\top \parallel \varphi$. As usual, we use the notation $D \vdash \sigma \stackrel{i}{\Longrightarrow}{}^* \sigma'$ for the transitive closure.

For every i-parameterized abstract machine there are three kinds of states σ_f for which no reduction is defined: $v \parallel \varphi$ corresponds to programs terminating in a value v with an output trace φ with maximum possible size i; $\texttt{error}(str) \parallel \varphi$ corresponds to programs terminating abnormally because of a runtime error $\texttt{error}(str)$ with trace output φ with maximum size i; and $\top \parallel \varphi$ corresponds to abnormally terminating programs due to a reduction step that would lead to an output trace larger than i. The latter case captures the finite prefix of size i of a program whose execution leads to an output trace of size larger than i.

Given an initial state σ_o of the form $e \parallel \varnothing$, and any natural number i, the behavior of the reductions $\stackrel{i}{\Longrightarrow}$ is well defined. We define the restricted evaluation function $eval^i$ for each one of the sequential machines in Figure 4. We extend the range of the evaluator with the extra element $\bot \parallel \varphi$ and we map all diverging programs with output trace φ to $\bot \parallel \varphi$. Since each machine admits only programs with maximum output trace i, even diverging programs have a finite output trace. We use $\varphi \sqsubseteq \varphi'$ for the standard prefix ordering on traces.

The full evaluator $eval$ for λ^{con} is defined in figure 4 in terms of the restricted evaluators for λ^{con}. When a program diverges we can define its output trace by using its finite prefixes obtained from the restricted evaluators.To show that evaluator $eval$ is meaningful, it suffices to prove it is a total function.

THEOREM 1. *eval is a total function.*

3. Concurrent Contract Checking

The language presented in the preceding section enables contract checking, but the abstract machines modeling its operational semantics are sequential. We now present an alternative semantics for a language with the same syntax (figure 1) that enables parallel contract checking. We call the resulting language λ^{ccc}. The semantics is described with a family of machines parameterized by a natural number i as in λ^{con}. As before a compilation step precedes the evaluation of a program.

The operational semantics of λ^{ccc} is concurrent and implements a master-slave architecture. The master thread evaluates the program and sends any contract it encounters in the context of an expression $\texttt{check}(e_1); e_2$ to the slave thread. The slave evaluates the contract predicate e_1 that it receives and aborts the program when a contract failure occurs. The two threads synchronize when effects, such as outputs, are performed or when errors are reported.

Again, we use two levels of reductions: local reductions—involving only expressions, from either the master or the slave threads—and global reductions—regulating the interaction of the master and the slave threads. The local evaluation relation is unchanged from λ^{con} (figure 2).

The states of λ^{ccc} are just like those of λ^{con}, except that we add a queue to manage communication between the master and slave threads. The initial state of program p is the state in which the queue is the empty queue $()$. A queue is a finite sequence of expressions. Two queues can be appended using **++**, defined in the usual way.

A global computation involves the entire state and requires the synchronization of the master and the slave thread. We use $D \vdash q \parallel e \parallel \varphi \stackrel{i}{\Longrightarrow}_s q' \parallel e' \parallel \varphi'$ to denote a global computation step from state $q \parallel e \parallel \varphi$ to state $q' \parallel e' \parallel \varphi'$. Figures 5 and 6 display the two sets of rules, which jointly specify global progress. The rules in figure 5 determine how the machine behaves when the queue is empty, that is, when the master may proceed independently of the slave. We have factored out the rules of figure 5 because we can reuse them for the correctness proof in a different context (τ). The rules in figure 6 focus on the case when the queue is not empty. They specify how local computations affect the global state and how reductions that depend on the global state are performed.

The master can delegate a task to the slave using rule (s-enq). This rule assigns a new meaning to the sequential operator $\texttt{check}(e_1); e_2$. The slave receives e_1 and adds it to the end of the queue. When e_1 reaches the front of the queue, the slave starts evaluating e_1. That way, the expressions in the queue are evaluated exactly in the same order they are added, which is the same order as a sequential version of the program would have used.

When the slave has finished evaluating an expression, the outcome of the reduction is simply discarded and the value is removed from the queue using rule (m-deq). This new semantics for $\texttt{check}(e_1); e_2$ ensures that obligations with a $\texttt{future}(c)$ contract ask the run-time system to parallelize the check of c, because those obligations reduce to `check` expressions.

To ensure that blame assignment and output traces are not affected by the nondeterministic nature of the parallel machine, we impose three kinds of synchronization points between the master and the slave: rules (m-err), (m-out) and (m-fout). The master cannot perform any effectful operation or signal an error or terminate

$$
\begin{array}{llllll}
D & \vdash & () \parallel E[e] \parallel \varphi & \overset{i}{\Longrightarrow}_\tau & () \parallel E[e'] \parallel \varphi & \text{where } D \vdash e \longrightarrow e' \quad \text{(m-prop)} \\
D & \vdash & () \parallel E[e] \parallel \varphi & \overset{i}{\Longrightarrow}_\tau & \texttt{error}(\texttt{"}x\texttt{"}) \parallel \varphi & \text{where } D \vdash e \longrightarrow \texttt{error}(\texttt{"}x\texttt{"}) \quad \text{(m-err)} \\
D & \vdash & () \parallel E[\texttt{check}(v); e] \parallel \varphi & \overset{i}{\Longrightarrow}_\tau & () \parallel E[e] \parallel \varphi & \quad \text{(m-check)} \\
D & \vdash & () \parallel E[\texttt{output}(v); e] \parallel \varphi & \overset{i}{\Longrightarrow}_\tau & () \parallel E[e] \parallel \varphi; v & \text{if } |\varphi| < i \quad \text{(m-out)} \\
D & \vdash & () \parallel E[\texttt{output}(v); e] \parallel \varphi & \overset{i}{\Longrightarrow}_\tau & \top \parallel \varphi & \text{if } |\varphi| = i \quad \text{(m-fout)} \\
D & \vdash & () \parallel E[v] \parallel \varphi & \overset{i}{\Longrightarrow}_\tau & v \parallel \varphi & \quad \text{(m-return)}
\end{array}
$$

Figure 5. Reduction rules for global evaluation with an empty queue, $\tau \in \{s, c\}$

$$
\begin{array}{llllll}
D & \vdash & e_q; q \parallel F[e] \parallel \varphi & \overset{i}{\Longrightarrow}_c & e_q; q \parallel F[e'] \parallel \varphi & \text{where } D \vdash e \longrightarrow e' \quad \text{(s-prop)} \\
D & \vdash & E[\texttt{output}(v); e_1]; q \parallel e \parallel \varphi & \overset{i}{\Longrightarrow}_c & E[e_1]; q \parallel e \parallel \varphi; v & \text{if } |\varphi| < i \quad \text{(m-qout)} \\
D & \vdash & E[\texttt{output}(v); e_1]; q \parallel e \parallel \varphi & \overset{i}{\Longrightarrow}_c & \top \parallel \varphi & \text{if } |\varphi| = i \quad \text{(m-qfout)} \\
D & \vdash & E[e_q]; q \parallel e \parallel \varphi & \overset{i}{\Longrightarrow}_c & E[e'_q]; q \parallel e \parallel \varphi & \text{where } D \vdash e_q \longrightarrow e'_q \quad \text{(m-qprop)} \\
D & \vdash & v; q \parallel e \parallel \varphi & \overset{i}{\Longrightarrow}_c & q \parallel e \parallel \varphi & \quad \text{(m-deq)} \\
D & \vdash & E[e_q]; q \parallel e \parallel \varphi & \overset{i}{\Longrightarrow}_c & \texttt{error}(\texttt{"}x\texttt{"}) \parallel \varphi & \text{where } D \vdash e_q \longrightarrow \texttt{error}(\texttt{"}x\texttt{"}) \quad \text{(m-qerr)} \\
D & \vdash & q \parallel F[\texttt{check}(e_1); e_2] \parallel \varphi & \overset{i}{\Longrightarrow}_c & q\texttt{++}e_1; () \parallel F[e_2] \parallel \varphi & \quad \text{(s-enq)} \\
D & \vdash & E[\texttt{check}(v); e_2]; q \parallel e \parallel \varphi & \overset{i}{\Longrightarrow}_c & E[e_2]; q \parallel e \parallel \varphi & \quad \text{(m-qcheck)}
\end{array}
$$

Figure 6. Reduction rules for global evaluation with a non empty queue

$$
eval^i_c(D \texttt{ in } e) = \begin{cases} \sigma_f & \text{if } D \vdash () \parallel e \parallel \varnothing \overset{i}{\Longrightarrow}^*_c \sigma_f, \\ \bot \parallel \varphi & \text{if } \forall e' \text{ such that } D \vdash () \parallel e \parallel \varnothing \overset{i}{\Longrightarrow}^*_c q' \parallel e' \parallel \varphi,\ \exists e'' \text{ such that } q' \parallel e' \parallel \varphi \overset{i}{\Longrightarrow}^{n,m}_c q'' \parallel e'' \parallel \varphi \\ & \text{where } n \text{ is the overall number of reductions, } m \text{ the number of } (\text{m-}\ldots) \text{ reductions in} \\ & \text{the corresponding transition sequence, and } n > 0,\ m > 0. \end{cases}
$$

$$
eval_c(D \texttt{ in } e) = \begin{cases} r & \text{if } \exists i \in \mathbb{N} \text{ such that } eval^i_c(D \texttt{ in } e) = r \text{ and } r \neq \top \parallel \varphi \\ \bot \parallel \hat{\varphi} & \text{if } \forall i \in \mathbb{N}\ eval^i_c(D \texttt{ in } e) = \top \parallel \varphi_i \text{ and } \hat{\varphi} \text{ is the smallest (possibly infinite) trace} \\ & \text{such that } \varphi_0 \sqsubseteq \varphi_1 \sqsubseteq \varphi_2 \cdots \sqsubseteq \hat{\varphi} \end{cases}
$$

Figure 7. λ^{ccc} evaluators

the program when the slave has a non-empty queue. If the slave has a non-empty queue, then there are expressions that in sequential mode would be evaluated before the synchronization point. In contrast, the slave can perform any output or error operation promptly, since it evaluates expressions with the same order that they would have been evaluated on purely sequential machine.

3.1 Examples

Despite the restricted form of parallelization introduced by our semantics and the synchronization points, the parallel machine remains nondeterministic and parallelizes contract checking.

Let h be the contract

$$\texttt{future}(\texttt{contract}(even?) \mapsto \texttt{contract}(even?))$$

and consider the program

$$\texttt{val rec } add1 : h = \lambda x.x + \ulcorner 1 \urcorner \texttt{ in } (add1\ \ulcorner 2 \urcorner)$$

The compilation process inserts obligations as follows:

$$\texttt{val rec } add1 : h = \lambda x.x + \ulcorner 1 \urcorner \texttt{ in } (add1^{h,\texttt{"}add1\texttt{"},\texttt{"}main\texttt{"}}\ \ulcorner 2 \urcorner)$$

and the result is then evaluated:

$$
\begin{array}{l}
() \parallel (add1^{h,\texttt{"}add1\texttt{"},\texttt{"}main\texttt{"}}\ \ulcorner 2 \urcorner) \parallel \varnothing \\
\Longrightarrow_c \\
\quad () \\
\quad \parallel \texttt{let } x = \\
\qquad \ulcorner 2 \urcorner^{\texttt{future(contract(}even?\texttt{))},\texttt{"}main\texttt{"},\texttt{"}add1\texttt{"}} \\
\qquad\quad \texttt{in let } y = \\
\qquad\qquad (add1\ x) \\
\qquad\qquad\quad \texttt{in } y^{\texttt{future(contract(}even?\texttt{))},\texttt{"}add1\texttt{"},\texttt{"}main\texttt{"}} \\
\quad \parallel \varnothing \\
\Longrightarrow_c \\
\quad () \\
\quad \parallel \texttt{let } x = \\
\qquad \texttt{check(if } (even?\ \ulcorner 2 \urcorner)\ \ulcorner 2 \urcorner\ \texttt{blame("}main\texttt{"))};\ulcorner 2 \urcorner \\
\qquad\quad \texttt{in let } y = \\
\qquad\qquad (add1\ x) \\
\qquad\qquad\quad \texttt{in } y^{\texttt{future(contract(}even?\texttt{))},\texttt{"}add1\texttt{"},\texttt{"}main\texttt{"}} \\
\quad \parallel \varnothing
\end{array}
$$

At this point, a sequential contract monitoring system would check the argument of $add1$ locally. The semantics of λ^{ccc} however delegates the contract checking to the slave. The master can continue with the evaluation of the successor of 2 in parallel:

$$
\begin{array}{l}
\Longrightarrow^*_c \\
\quad \texttt{if } (even?\ulcorner 2 \urcorner\ )\ulcorner 2 \urcorner\ \texttt{blame("}main\texttt{")};() \\
\quad \parallel \ulcorner 3 \urcorner^{\texttt{future(contract(}even?\texttt{))},\texttt{"}add1\texttt{"},\texttt{"}main\texttt{"}} \\
\quad \parallel \varnothing \\
\Longrightarrow^*_c
\end{array}
$$

$$
\begin{array}{l}
\quad \texttt{if}\ (even?\ \ulcorner 2 \urcorner)\ \ulcorner 2 \urcorner\ \texttt{blame}(\texttt{"}main\texttt{"}); () \\
\quad \| \texttt{check}(\texttt{if}\ (even?\ \ulcorner 3 \urcorner)\ \ulcorner 3 \urcorner\ \texttt{blame}(\texttt{"}add1\texttt{"})); \ulcorner 3 \urcorner \\
\quad \| \varnothing \\
\Longrightarrow_c^* \\
\quad \texttt{if}\ (even?\ \ulcorner 2 \urcorner)\ \ulcorner 2 \urcorner\ \texttt{blame}(\texttt{"}main\texttt{"}); \\
\qquad \texttt{if}\ (even?\ \ulcorner 3 \urcorner)\ \ulcorner 3 \urcorner\ \texttt{blame}(\texttt{"}add1\texttt{"}); () \| \ulcorner 3 \urcorner \| \varnothing \\
\Longrightarrow_c^* \\
\quad \texttt{true}; \texttt{if}\ (even?\ \ulcorner 3 \urcorner)\ \ulcorner 3 \urcorner\ \texttt{blame}(\texttt{"}add1\texttt{"}); () \| \ulcorner 3 \urcorner \| \varnothing \\
\Longrightarrow_c^* \\
\quad \texttt{blame}(\texttt{"}add1\texttt{"}); () \| \ulcorner 3 \urcorner \| \varnothing
\end{array}
$$

which yields the final result $\texttt{error}(\texttt{"}add1\texttt{"}) \| \varnothing$.

As this example shows, the master and the slave must collaborate. In particular, the master cannot ignore the will of the slave once the slave is initiated. The slave is the one that controls the execution of effectful operations. Moreover, the master is not even allowed to diverge without the permission of its slave.

To illustrate these points, we turn to a couple more examples. First, let h be the following contract

$$\texttt{future}(\texttt{contract}(\lambda x.\texttt{false}) \mapsto \texttt{contract}(\lambda x.\texttt{false})))$$

in the following program

$$\texttt{val rec}\ loop : h = \lambda x.(loop\ \ x + \ulcorner 1 \urcorner)\ \texttt{in}\ (loop\ \ulcorner 0 \urcorner)$$

If the program is executed sequentially, then the domain contract checking fails without calling $loop$. In contrast the parallel machine may choose to send the contract to the slave and then ignore the slave for the following steps, start evaluating the $loop$ call and diverge. Such an execution disagrees with the sequential evaluation and should be excluded from the set of valid evaluations. We therefore restrict valid diverging executions to those that let the slave take steps infinitely often. This way the slave becomes the guard of sequentiality and the master of nondeterminism.

Our semantics enforces the above restriction by requiring that valid transition sequences take steps described by *mandatory* rules (m-...) infinitely often. We use the notation $\sigma \Longrightarrow_c^{n,m} \sigma'$ to indicate that σ reduces to σ' in n steps out of which m are mandatory.

Consider now the following subtle scenario. Let h be the following contract

$$\texttt{future}(\texttt{contract}(hostile?)) \mapsto \texttt{contract}(\lambda x.\texttt{true})$$

in the following program

$$
\begin{array}{l}
\texttt{val rec}\ attack : h = \\
\quad \lambda target.\texttt{output}(\texttt{"}missile\texttt{"}); \texttt{"}mission\ accomplished\texttt{"} \\
\texttt{in}\ (attack\ \ \texttt{"}ally\texttt{"})
\end{array}
$$

When contracts are checked in-line, the domain contract of $attack$ ensures that a missile is fired only if the target is hostile. However, a parallel and unsynchronized check of the contract may lead to missile launching without first confirming that the target is an enemy. This indicates that observable effects should be delayed until all parallel contract checking has completed. Our semantics accomplishes this by treating the effectful operators as synchronization points between the master and the slave.

3.2 Determinism and Correctness

We define the restricted parallel evaluators $eval_c^i$ in figure 7 in the same way as we defined $eval^i$ for λ^{con}.

To establish that evaluators $eval_c^i$ are functions, we need to show that all possible executions of a program in the parallel machine have the same observable behavior. More precisely, we must establish that all possible executions of a program have the same transition sequence output and termination behavior. We prove this fact in the traditional way using an extended form of the Diamond Lemma for our language (Flanagan and Felleisen 1999).

LEMMA 2. *For all $i \in \mathbb{N}$, if $D \vdash \sigma \overset{i}{\Longrightarrow}_c^* \sigma_f$ then*

(a) if $D \vdash \sigma \overset{i}{\Longrightarrow}_c^* \sigma_f'$, $\sigma_f = \sigma_f'$

(b) there is no transition such that $\forall k.\ D \vdash \sigma \overset{i}{\Longrightarrow}_c^* \sigma_k$, $m_k > 0$ and $D \vdash \sigma_k \overset{i}{\Longrightarrow}_c^{n_k,m_k} \sigma_{k+1}$.

Lemma 2 can be used to show that evaluators $eval_c^i$ are functions. We define the evaluator $eval_c$ for our language in figure 7 and we show that $eval_c$ is a total function.

THEOREM 3. *$eval_c$ is a total function.*

Our ultimate goal, though, is to show that the parallel language λ^{ccc} is semantically equal to λ^{con}. In other words, we would like to prove that the two evaluators $eval_c$ and $eval$ are equal.

The differences between the two languages makes it difficult to compare them. To overcome the problem, we define an intermediary language that bridges the gap between λ^{con} and λ^{ccc}. It is sequential like λ^{con}, but shares the same machine infrastructure with λ^{ccc}.

The semantics of this new language λ_{seq}^{ccc} is described in figures 2 and 5. We define its restricted sequential evaluators $eval_s^i$ and $eval_s$ in figure 8.

LEMMA 4. *$eval_s$ is a total function.*

Now that we have established that $eval_s$ is a well-defined total function, we can easily prove that the two sequential languages λ^{con} and λ_{seq}^{ccc} are equivalent.

LEMMA 5. *$eval_s = eval$.*

The next step is to show that λ_{seq}^{ccc} is also equivalent to λ^{ccc}. A first result is that the parallel language is only an extension of the sequential language. Thus, the parallel language can admit all the transition sequences that the sequential machine admits.

LEMMA 6. *For all $i \in \mathbb{N}$, if $D \vdash \sigma \overset{i}{\Longrightarrow}_s \sigma'$ then $D \vdash \sigma \overset{i}{\Longrightarrow}_c \sigma'$.*

Now we show that $eval_c$ is equal to $eval_s$ and thus we establish that the two corresponding models are equivalent.

LEMMA 7. *$eval_c = eval_s$.*

Lemmas 5 and 7 imply that λ^{con} and λ^{ccc} are equivalent.

THEOREM 8. *$eval = eval_c$.*

Equipping our system with multiple slaves that evaluate the elements of the queue in parallel and adapting the correctness proof to the new conditions is rather straightforward.

4. Effects Beyond Outputs

Extending our model with mutable reference cells is straightforward. It suffices to add expressions for cell allocation plus operators for reading from and writing to reference cells. Reading and writing must be treated as synchronization points between the slave and the master. The master can write and read a location in the store only if the queue is empty.

Adding a store does not significantly affect our model. It already deals successfully with outputs and the corresponding output trace. The store is often treated in a way that is simpler than outputs because when a program diverges, its store is irrelevant; its output trace, in contrast, is part of its observable behavior. An output is an irreversible observable event, while a store update can be masked by a subsequent update. The store can therefore be seen as a restricted form of an output trace. The proof technique that

$$eval_s^i(D\ \texttt{in}\ e) = \begin{cases} \sigma_f & \text{if } D \vdash () \parallel e \parallel \varnothing \stackrel{i}{\Longrightarrow}_s^* \sigma_f \\ \bot \parallel \varphi & \text{if } \forall e' \text{ such that } D \vdash () \parallel e \parallel \varnothing \stackrel{i}{\Longrightarrow}_s^* () \parallel e' \parallel \varphi,\ \exists e'' \text{ such that } D \vdash () \parallel e' \parallel \varphi \stackrel{i}{\Longrightarrow}_s () \parallel e'' \parallel \varphi \end{cases}$$

$$eval_s(D\ \texttt{in}\ e) = \begin{cases} r & \text{if } \exists i \in \mathbb{N} \text{ such that } eval_s^i(D\ \texttt{in}\ e) = r \text{ and } r \neq \top \parallel \varphi \\ \bot \parallel \hat{\varphi} & \text{if } \forall i \in \mathbb{N}\ eval_s^i(D\ \texttt{in}\ e) = \top \parallel \varphi_i \text{ and } \hat{\varphi} \text{ is the smallest (possibly infinite) trace such} \\ & \text{that } \varphi_0 \sqsubseteq \varphi_1 \sqsubseteq \varphi_2 \sqsubseteq \cdots \sqsubseteq \hat{\varphi} \end{cases}$$

Figure 8. λ_{seq}^{ccc} evaluators

we use to show our approach's correctness is sufficient to cover a modification of our language with reference cells.

Moreover, the addition of a store affects the practicality of our approach. A naive addition can lead to an increase in the number of synchronization points. Synchronization points between threads are an important factor for performance loss in concurrent programs. But the nature of synchronization due to operations on locations are different than those due to output operations. Operations on different locations do not need to be synchronized. A write operation on a location performed by the master thread needs to wait only for writes and reads on the same location performed by the contracts that in a sequential execution would be checked before the write operation. This implies that if some operations involve only locations that are not manipulated by such contracts then those operations should not be treated as synchronization points. An approximation algorithm to identify operations with this property can be derived from standard control flow static analysis techniques (Shivers 1991). Using the control graph of a program we can collect the set of contracts that are executed before any given operation on the store. Using the data flow graph of a program we can conservatively determine which locations are reachable (Dimoulas and Wand 2009) from the variables of a contract predicate and the set of locations reachable from the arguments of the given operation on the store. With this information, we can conclude that if this set of locations has an empty intersection with the set of locations that are reachable from the variables of the contracts, the operation does not need to be a synchronization point. As an immediate consequence, store operations do not need to be synchronization points when contracts are purely functional.

The above reasoning, however, is applicable only within the limited scope of our model. In a realistic setting, where contract may throw exceptions that can be caught by the program, which then proceeds to recovery, the contents of the store upon contract failure cannot be ignored. This turns every cell access into a synchronization event.

In functional programming where effects are infrequent, the cost of synchronization is not an important factor. However, in imperative programming where almost every other operation involves access to the store, synchronization becomes a bottleneck for the system's performance. The development of sophisticated and accurate effect analysis then becomes an emerging necessity.

5. Implementation

The language λ^{con} is a model of PLT Scheme and its contract library. From that perspective, λ^{ccc} can be viewed as a compact specification for a parallel re-implementation of the PLT contract library. In this section, we explain how to implement λ^{ccc} for a full-fledged system.

PLT Scheme (Flatt 2009) is a full-featured higher-order language with effects and comes with a sophisticated contract system. Contracts are implemented as error projections (Findler and Blume 2006) in an autonomous macro library. This organization facilitates changes to the contract monitoring mechanism by limiting those changes to the library.

Although the released version of PLT Scheme does not support multi-core programming, an experimental version offers `places`,[1] a prototype extension for multi-core programming. Our system is bringing together the contract system of PLT Scheme and `places`.

We use `places` to launch two operating system threads that map to the two threads of the λ^{ccc} model: the master and the slave. Following our model, communication between the two parties is implemented through a shared queue.

Before a program's evaluation begins in the master thread, the master initiates the slave. The slave starts listening to the queue, waiting until it becomes non-empty. The master evaluates the program and every time it hits a flat future contract it adds it to the end of the queue. When a contract is wrapped with our new `future/c` combinator, the resulting projection is shipped to the queue as a closure. The slave picks the transmitted contract and evaluates it. If it fails, it aborts the system with a contract error message. Otherwise it removes the contract from the queue and waits for the next one. In the meantime, the master continues the evaluation of the rest of the program. In case the master needs to perform an output or the evaluation terminates, the master waits for all the elements of the queue to be processed and then proceeds with the output statement or terminates.

The central piece of our prototype is an efficient implementation of the queue, shown in figure 9. Our implementation is based on the algorithm suggested by Herlihy and Shavit (2008, paragraph 10.3).

In order to ensure that access to the queue does not become a bottleneck in practice, the queue is a mutable list with two pointers: one to the first and one to the last element. The head end is responsible for reading and dequeuing elements from the list and the tail end is responsible for enqueuing new elements. In this setting, as long as the enqueue and dequeue operations are applied to different locations, locks are needed only to internally synchronize groups of readers and writers (Herlihy and Shavit 2008, paragraph 10.3). To make sure that enqueue and dequeue are never performed on the same element, we use a sentinel element to prevent the queue from ever becoming empty after the first enqueue.

Our system uses only two threads: the master which controls the tail of the queue and performs enqueues and the slave which controls the head and performs peeks and dequeues. We therefore do not require any locks in our implementation, which simplifies the original algorithm.

The queue is initially empty and both pointers point to empty lists. The first time an element is added to the queue the head and tail pointers are updated to point both to this first element. No dequeues are performed prior to this point as the slave is waiting for the queue to become non-empty. When more elements are added, the tail pointer is updated to point to the last element of the queue.

[1] The prototype for `places` is being developed by Kevin Tew and Matthew Flatt at the University of Utah.

```
    #lang scheme                                                    ;; the queue module

    (require scheme/mpair)

    (define-struct node (content [done? #:mutable]))                ;; a node contains the thunk content that
                                                                    ;; returns any kind of scheme values (the
                                                                    ;; closure of a contract projection)and the
                                                                    ;; boolean flag done?
                                                                    ;; the done? flag is #t if the node has been
                                                                    ;; already visited

    (define-struct queue (head tail)                                ;; a queue is a list of nodes with  mutable
                   #:mutable)                                       ;; pointers at the first and last element

    (define QUEUE  (make-queue null null))                          ;; initially the queue does not contain any
                                                                    ;; nodes
                                                                    ;; this holds only before the first enqueue

    (define (is-queue-mt?)                                          ;; a queue is empty
      (or (null? (queue-tail QUEUE))                                ;; if it has no nodes
          (node-done? (mcar (queue-tail QUEUE)))))                  ;; or it only has a sentinel node

    (define (dequeue)                                               ;; the tail node is removed if the queue has
      (cond ((null? (mcdr (queue-head QUEUE)))                      ;; more than one nodes
             (set-node-done?! (mcar (queue-head QUEUE)) #t))        ;; if the queue has only one node then the
            (else                                                   ;; node is marked as visited but it is not
             (set-queue-head! QUEUE (mcdr (queue-head QUEUE))))))   ;; removed (sentinel node)

    (define (enqueue closure)                                       ;; new not visited nodes are added to the tail
      (let ((tail (queue-tail QUEUE)))
        (cond ((null? tail)
               (let ((head-tail (mcons (make-node closure #f) null)))
                 (set-queue-head! QUEUE head-tail)
                 (set-queue-tail! QUEUE head-tail)))
              (else
                (begin
                  (set-mcdr! tail (mcons (make-node closure #f) null))
                  (set-queue-tail! QUEUE (mcdr tail)))))))

    (define (peek-queue)                                            ;; the contents of the unvisited head node
      (cond ((null? (queue-head QUEUE)) 'empty)                     ;; are returned but the node is not removed
            ((null? (mcdr (queue-head QUEUE)))                      ;; from the queue
             (if (node-done? (mcar (queue-head QUEUE)))
               'empty
               (node-content (mcar (queue-head QUEUE)))))
            (else
              (if (node-done? (mcar (queue-head QUEUE)))            ;; enqueue may push a visited node at the
                (begin                                              ;; head position
                  (set-queue-head! QUEUE (mcdr (queue-head QUEUE))) ;; the visited node is removed if the queue
                  'empty)                                           ;; has more than one nodes
                (node-content (mcar (queue-head QUEUE)))))))

;;-------------------------------------------------------------------------
;;---------------------------------- provide ------------------------------

    (provide/contract
      [struct node  ((content (->   any/c)) (done? boolean?))]
      [struct queue ((head (mlistof node?)) (tail (mlistof node?)))]
      [is-queue-mt? (->   boolean?)]
      [enqueue      (-> (->   any/c) void?)]
      [dequeue      (->   void?)]
      [peek-queue   (->   (or/c (->   any/c) symbol?))])
```

Figure 9. The queue implementation

We use an extra tag for each element that indicates if the element has been visited by the slave. When an element is added to the end of the queue through its tail pointer, the tag is set to a NOTVISITED state. This way we mark checked contracts and make sure that the content of a node is delivered to the slave at most once. Contracts are not delivered to the slave only when the slave has aborted the program because it discovered a contract violation.

Access to the elements from the head of the queue is performed in two steps. First the element is picked for processing, using `peek-queue`, without being removed from the queue. Second, when the first element is processed, the element is removed from the queue and the head pointer is moved to the next element, if the queue has more than one element. Otherwise, both pointers point to the same element. In this case, `dequeue` does not removes the element from the queue but sets its tag to VISITED. Thus we avoid having an empty queue and `dequeue` and `enqueue` never access destructively the same end of the queue.

Visited elements may appear at the head of the queue even when the queue contains more than one element. This case arises when a number of enqueues are performed on a single-element queue. When the `peek-queue` operation finds a VISITED element at the head of the queue and the queue has more than one element, it removes the element and moves the head pointer to the next element. Because this operation can occur only if the queue has two or more elements, the algorithm guarantees that removal and additions to the queue are never applied to the same end.

To implement synchronization at effectful operations, we use wrapper functions that inspect the queue. They are inserted to the program's code manually. A real implementation would replace these wrappers with hooks into the PLT Scheme compiler. The compiler should be notified that parallel contract checking is enabled and instrument all effectful operations, error reports and program termination with automatic synchronization checks.

6. Evaluation

6.1 Benchmarks

While benchmarks for Scheme and PLT Scheme abound, there are no benchmark suited for evaluating the performance of contract monitoring. In the past Findler has measured the performance of the contract system of DrScheme (Findler et al. 2002), a 200 kloc program, but this is not feasible with a prototype implementation of the contract library.

We have therefore developed a small suite of benchmarks from the existing test suite of PLT Scheme[2] and from extracts of small applications. For all these benchmarks we turned informal assertions in comments into formal contracts. The architecture of our benchmarks follow a simple idea: a set of libraries provide functionality that is guarded by contracts. The client module imports all the libraries and makes extensive use of the provided functions. This design corresponds to what we think is a rough sketch of a realistic and large application with interesting contracts.

Here is a detailed description of the benchmarks, with a focus on the contract computations that they perform:

fber (192 lines in 4 modules) is an interpreter for Boolean programs, that is, Boolean expressions and functions. The interpreter's contract ensures that the given expressions and functions are closed and well-formed. Similar properties are checked by the contract of the substitution function. The input to the benchmark consists of a Boolean expression with 10000 leaves that is applied to a function whose body has also 10000 leaves.

dna (241 lines in 3 modules) simulates the attack of a virus group on a cell's colony. A cell is represented as two complementary DNA chains. A virus consists of a DNA sequence that stands for the target DNA sequence for the virus and a mutated DNA sequence that replaces the target DNA sequence when the virus attacks a cell. The latter possibly contains two special complementary bases X and Y. The mutation function's contract checks if the target cell and the attacker virus are well-formed according to the above description. The input to the benchmark consists of a group of 50 viruses of DNA length 200 and a colony of 200 cells of DNA length 500. All the viruses attack all the cells of the colony.

lab (258 lines in 3 modules) is a variant of the `dna` benchmark. In this scenario, we have two cell colonies. All the viruses of the group of viruses attack all the cells of the first colony. The second colony manages to survive an attack by one of the viruses and reproduces twice before the virus attacks again. After that the mutated cells die and then the virus performs its last attack. All the functions have simple type contracts except for one, which comes with a contract that checks if the attacked cell is well-formed and is used for the last two attacks. The input to the benchmark consists of a group of 50 viruses of DNA length 200 and two identical colonies of 200 cells of DNA length 500.

tel (356 lines in 3 modules) implements a phone book. Each entry consists of a last name, a first name, a list of telephone numbers with annotations, and a list of email addresses (also with annotations). A phone book is a sorted list of entries. The contracts of the `add` and `remove` functions check if the given telephone catalog is well-formed. The input to the benchmark is a catalog of 100 entries, on which 2000 add and remove operations are performed.

tel-bst (417 lines in 3 modules) re-implements the previous phone book as an AVL tree. The contracts of the `add` and `search` functions check if the given telephone catalog is a well-formed AVL tree. The input to the benchmark is a catalog of 1000 entries, on which 2000 search and add operations are performed.

lists (59 lines in 3 modules) provides a function `min` to find the minimum element of a given sorted list of positive numbers and a function `incr` that increments all the elements of the sorted list by the minimum element of the list. The `min` function simply returns the first element of the list if the list is not empty and an empty-list symbol otherwise. The contracts of the functions check if the given lists are sorted lists of positive numbers. The input to the benchmark is a list of 500,000 elements.

conform (702 lines in 8 modules) turns a graph into a lattice. The application comes with a sets manipulation library. A set is represented as a list of distinct elements. The contracts on the functions of the library check if the set arguments are properly formed. The input to the benchmark is a graph with four nodes and six edges. The application is called ten times. The benchmark fails to run faster with `future` contracts because the arguments to the set library functions are small and evaluating the related contracts in parallel does not cover for the communication and synchronization cost.

graphs (731 lines in 6 modules) produces all directed graphs with n nodes, distinguished root and out-degree bounded by 2, up to isomorphism. The benchmark's functions come with a number of assertions that were turned into contracts of the exported functions. Almost all of them concern checking of type or other simple structural properties of the arguments and the results of functions.The input to the benchmark is a graph with 6 nodes. and returns 44 non-isomorphic graphs. The application is called

[2] `http://svn.plt-scheme.org/plt/trunk/collects/tests/mzscheme/benchmarks/common/`

three times. The benchmark fails to produce speedup with parallel contract execution because the low complexity of its contracts do not cover for the communication and synchronization cost.

div (100 lines in 4 modules) partitions lists of even lengths into two equal halves. Two functions are provided: a recursive and an iterative variant. Each function comes with a contract that checks if the given list has even length. The input to the benchmark is a 200 elements list and the test loop performs 240,000 divisions.

The last three benchmarks originate from the PLT benchmarks set.

6.2 Measurements

For our experiments we used a Dell Latitude D820 laptop with a Core Duo T2400 CPU @ 1.83 GHz, 667MHz front side bus, 2MB L2 cache, 2 GB RAM memory and Ubuntu 7.10 operating system.

For our evaluation, we conducted two experiments. Our first experiment simulates an un-critical programmer who randomly annotates contracts with future. Specifically, our experiment setup evaluates each benchmark with randomly chosen half of contracts turned into `future` contracts. We made five such random annotations and analyzed their average results. The analysis suggests some basic conjectures about contract evaluation; most importantly, only contracts whose cost competes with the cost of the function body (up to synchronization points) should be annotated with `future/c`. In our second experiment we confirmed these conjectures by carefully selecting `future` contracts and repeating the measurements.

Figure 10 shows the results of the first experiment. The use of `future/c` leads to speedup in some cases but not always. There are two distinct reasons for a lack of speedup:

- The contract's asymptotic complexity is less than the complexity of communication. This is examplified by the **graphs** benchmark, in which the majority of contracts are type-like and shallow structure-properties contracts.
- The contracts asymptotic complexity is large but the inputs are so small that the cost of contract checking does not cover the communication cost. This is exemplified by the **conform** benchmark, in which all set operations are applied to sets with fewer than ten elements.

The effect of low-cost future contracts on the execution time of a program depends on the number of calls of the functions that these contracts guard. If a small number of low-cost contracts is sent to the slave then the overhead of communication is insignificant. For instance for the **tel** benchmark this accumulated cost is less that 200msec and does not manage to cancel the speedup caused by more complex contracts. When the number of transmitted low-cost contracts increases however the accumulated cost affects the speedup of the system. The **lab**, **conform** and **graphs** benchmarks demonstrate this point; 50% of their contracts guard functions that are used scarcely, so when they are transformed to `future` contracts they don't produce significant change of the execution time; while the other 50% protect commonly used functions and annotating them with `future` contracts, causes the program execution to slow down. Similarly, complex contracts that guard functions that are called many times contribute much more to the reduction of the execution time than contracts for functions that are rarely used.

The **tel-bst** and **lists** benchmarks reveal another subtle issue. When this benchmark is executed sequentially, most of the execution time is spent evaluating the contracts rather than the calls of the functions they guard. More specifically 50% of the contracts of the benchmark are much more complex asymptotically than the bodies of the functions they protect. Annotating these contracts as `future` contracts leads to saturation of the slave. Contract checking dominates execution and the master remains idle for the last part of the evaluation. Any contracts that reside at the queue at this point contribute only to the accumulated communication cost and they do not improve the performance of the system. Practically speaking they are not checked in parallel.

A system with multiple slaves could possible handle extraordinary long queues and the saturation issue. However, multiple threads cannot lead to a linear improvement of the system's performance. The communication cost leads to a significant slowdown that more concurrency cannot cover for. Every light-weigth contract sent to the slave(s) delays the master thread more than its inline checking. In programs with many such contracts the accumulated delay becomes an issue.

This is the case of our **lab** benchmark. The application comes with 6 functions guarded by contracts. Figure 11 shows the speedup we get when annotating different number of contracts as `future` contracts. All the contracts are simple contracts that capture shallow structural properties, except for one that has a high asymptotic complexity. Also the function guarded by the latter contract is applied to large lists and these lists are also the arguments passed to the contract. When only this contract is turned into a `future` contract, the speedup increases significantly. A naive approach to parallel contract monitoring would lead a programmer to annotate even more contracts. However, the contracts left without `future` annotation are inappropriate for parallel checking The delegation of their evaluation to the slave results in gradual but significant slowdown of the system.

Figure 12 presents the results of the second experiment. For this second experiment, we carefully chose contracts for parallel evaluation according to their relative cost. Specifically, we annotated the contracts in the order that they best satisfy our criteria. In every case, when 50% percent of the contracts of a benchmark are annotated as `future` contracts and these contracts are selected carefully according to our conjectures rather than randomly, the system's performance improves compared to the the random selection.

Our results imply that the effective use of `future` contracts demands a careful analysis of each specific application. Naive strategies not only fail to lead to increase of speedup but also can impose a heavy time penalty. As expected, when the overhead of checking a contract is more than the communication cost, it is worth checking the contract in parallel with the rest of the code. If a program comes with complex contracts then annotating these contracts with `future` tends to result in a significant improvement of the performance of the program. Parallel contract checking is most effective when the complexity of the contract of a function is comparable to the complexity of its computation up to the first effect statement. Also, note that the overhead from synchronization on output statements is not important. For example, in the **conform** benchmark the additional cost of output synchronization is less than 200ms. In general, wise use of the `future` annotation is highly beneficial as it encourages programmers to add good contracts to their functions obtaining more guarantees about their programs and restricting at the same time the penalty of contract monitoring.

7. Related Work

Two separate sources inspired our work: the runtime verification community (Avgustinov et al. 2007; Barnett et al. 2004; Chen and Roşu 2007; Havelund and Roşu 2001; Kim et al. 2004; Leavens et al. 2000; Zee et al. 2007) and work on the original `future` (Halstead 1984) construct of LISP and later Scheme.

Roughly speaking, run-time verification is analogous to monitoring a patient's status during surgery. A run-time verification monitor inserts instrumentation code into the program, and the instrumentation code sends the value of variables (of "flat" or serializable type) to a parallel "monitor thread." Over time these trans-

	0% future/c*	random 50% future/c*	random 50% future/c speedup	100% future/c*	100% future/c speedup
fber	18298	17565	1.04	16.328	1.12
dna	63042	47516	1.39	41970	1.50
lab	87996	90545	0.97	88632	0.99
tel	25318	22974	1.10	21443	1.18
tel-bst	32659	20195	1.62	27758	1.18
lists	14321	12151	1.20	12814	1.12
conform	10490	11324	0.95	12169	0.89
graphs	26343	28868	0.92	32892	0.80
div	13264	110363	1.20	87014	1.52

*average real time of five executions in msec

Figure 10. Experimental results for random annotations

	1/6 future/c	2/6 future/c	3/6 future/c	4/6 future/c	5/6 future/c	6/6 future/c
lab	1.07	1.05	1.00	0.99	0.98	0.99

Figure 11. The **lab** benchmark experimental results (speedup)

	random 50% future/c speedup	selected 50% future/c speedup
fber	1.04	1.07
dna	1.39	1.78
lab	0.97	1.06
tel	1.10	1.18
tel-bst	1.62	1.62
lists	1.20	1.46
conform	0.95	0.97
graphs	0.92	0.99
div	1.20	1.20

Figure 12. Comparison of experimental results between random and selected annotations

missions create an execution trace, which the monitor continuously inspects for externally specified logical properties. These specifications tend to be defined in some variant of temporal logic (Pnueli 1977). When a trace does not satisfy a specification, the run-time verification system either issues a warning or raises an exception.

In contrast to ordinary contract systems, a run-time verification system makes no attempt to synchronize the evaluation of the main program and the monitoring thread. When a violation is discovered, the main program may have progressed far beyond the point where the violation took place; in particular, it may have already issued damaging outputs, which a synchronized discovery of the violation would have prevented. Furthermore, in contrast to future contracts, run-time verification systems never in-line checks into the main program; after all, they monitor on a parallel thread only for a separation of concerns. Our contract system distributes contract monitoring over both the master and the slave thread.

Halstead's (Halstead 1984) `future` construct attempts to improve the performance of functional programs via parallel computations. Expressions wrapped in `future` are evaluated in parallel to the rest of the program. In their place, special placeholder values are injected into the main computation. When the computation applies a strict computational construct (`+`, `if`) to such placeholder values, the main program waits for the appropriate auxiliary thread to complete its computation. Our work borrows from Flanagan and Felleisen's semantic framework (Flanagan and Felleisen 1999) for Halstead's `future` construct.

Since contracts are usually functional expressions, and on occasion are written in special-purpose declarative notations even in imperative languages, the idea of adapting `future` from functional programming to contract programming is natural. The major difference between Halstead's `futures` and ours concerns synchronization. Our future contracts do not create placeholder values, because the result of contract expressions is never needed to evaluate the rest of the program. Synchronization for future contracts is inserted via a compiler or macro expander at effectful sites and is enforced at the end of the master and slave computations to preserve the meaning of loops.

8. Conclusion

This paper describes a step toward the parallelization of a contract monitoring system for higher-order languages. We introduce the notion of `future` contracts. In the same spirit as the `future` (Halstead 1984) construct, we take advantage of the implicit parallelism of functional languages to check `future` contracts in parallel with

the rest of the program. We prove that annotating a program's contracts with our `future` contract combinator does not change the semantics of the program even when the program uses effectful constructs like output statements and ref cells operators. We describe a prototype implementation of our system and through a series of benchmarks we show that our technique can speedup the execution of programs with contracts.

In the near future, we wish to consider two independent extensions of this work, with an eye towards a practical and efficient implementation of parallel contract checking. First, we intend to investigate how to take advantage of more than one monitoring thread. It is not clear how to do this efficiently, because of the additional challenges created by the need for synchronizing between those monitoring threads to respect sequential blame assignment. Second, we intend to develop an effect analysis to eliminate synchronization points and study the inclusion of additional effectful constructs such as exceptions in our system.

Both of these extensions will require us to look into existing work on the efficient implementation of `future` in parallel versions of Scheme and Lisp for shared-memory multiprocessors (Feeley 1993) as we expect that some of those techniques will apply in our setting. The translation is not immediate—future contracts are not future computations—but adaptation seems a possibility.

References

P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 589–608, 2007.

M. Barnett, K. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004, LNCS vol. 3362, Springer*, 2004.

F. Chen and G. Roşu. Mop: an efficient and generic runtime verification framework. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 569–588, 2007.

C. Dimoulas and M. Wand. The aggregate update problem. In *International Conference on Verification, Model Checking and Abstract Interpretation*, pages 44–58, 2009.

A. Duncan and U. Hoelzle. Adding Contracts to Java with Handshake. Technical report, Santa Barbara, CA, USA, 1998.

M. Feeley. *An efficient and general implementation of futures on large scale shared-memory multiprocessors*. PhD thesis, Brandeis University, 1993.

M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, pages 235–271, 1992.

R. Findler and M. Blume. Contracts as Pairs of Projections. In *International Symposium in Functional and Logic Programming*, pages 226–241, 2006.

R. Findler and M. Felleisen. Contracts for higher-order functions. In *ACM SIGPLAN International Conference on Functional Programming*, pages 48–59, 2002.

R. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for scheme. *Journal of Functional Programming*, 12: 369–388, 2002.

R. Findler, S. Guo, and A. Rogers. Lazy contract checking for immutable data structures. In *International Symposium on Implementation and Application of Functional Languages*, pages 22–34, 2007.

C. Flanagan and M. Felleisen. The semantics of future and an application. *Journal of Functional Programming*, 9(1):1–31, 1999.

M. Flatt. PLT MzScheme: Language manual. Technical Report PLT-TR2009-reference-v4.1.4, PLT Scheme Inc., 2009. `http://www.plt-scheme.org/techreports/`.

B. Gomes, D. Stoutamire, B. Vaysman, and H. Klawitter. A Language Manual for Sather 1.1, 1996.

R. Halstead. Implementation of Multilisp: Lisp on a multiprocessor. In *ACM Symposium on LISP and Functional Programming*, pages 9–17, 1984.

K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. Technical report, 2001.

M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

R. Holt, P. Matthews, J. Rosselet, and J. Cordy. *The Turing programming language: design and definition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

M. Karaorman, U. Holzle, and J. Bruno. jContractor: A Reflective Java Library to Support Design by Contract. Technical report, Santa Barbara, CA, USA, 1999.

M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-Mac: A Run-Time Assurance Approach for Java Programs. *Formal Methods in Systems Design*, 24(2):129–155, 2004.

M. Kölling and J. Rosenberg. Blue:Language Specification, version 0.94, 1997.

R. Kramer. iContract - The Java(tm) design by Contract(tm) Tool. In *Technology of Object-Oriented Languages and Systems*, page 295, 1998.

G. Leavens, K. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *Object-Oriented Programming, Systems, Languages, and Applications Companion*, pages 105–106, 2000.

D. Luckham and F. Von Henke. An Overview of Anna, a Specification Language for Ada. *IEEE Software*, 2(2):9–22, 1985.

B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.

R. Ploesch and J. Pichler. Contracts: From Analysis to C++ Implementation. In *Technology of Object-Oriented Languages and Systems*, page 248, 1999.

A. Pnueli. The temporal logic of programs. In *Symposium on the Foundations of Computer Science*, pages 46–57, Providence, Rhode Island, 1977.

J. Reppy. *Concurrent programming in ML*. Cambridge University Press, New York, NY, USA, 1999. ISBN 0-521-48089-2.

D. Rosenblum. A Practical Approach to Programming With Assertions. *IEEE Transactions on Software Engineering*, 21(1): 19–31, 1995.

O. Shivers. *Control-Flow Analysis of Higher-Order Languages, or Taming Lambda*. PhD thesis, Carnegie Mellon University, 1991.

K. Zee, V. Kuncak, and M. Rinard. Runtime Checking for Program Verification Systems. In *Runtime Verification*, pages 202–213, 2007.