Parallel Proof Checking in Isabelle/Isar

Makarius Wenzel

Technische Universität München
Institut für Informatik, Boltzmannstraße 3, 85748 Garching, Germany
`http://www.in.tum.de/~wenzelm/`

Supported by BMBF project "Verisoft" (grant 01 IS C38)

Abstract

We address the "multicore problem" for mathematical assistants with full proof checking, with special focus on Isabelle/Isar and its main SML platform Poly/ML. On the one hand, working with explicit definitions, statements, and proofs requires significant runtime resources, so the question of parallel checking is really relevant. On the other hand, the inherent structure of formal theories provides various possibilities for parallelism (both implicit and explicit), which is in fact an almost ideal situation. Exploiting this potential in practice requires to reconsider various aspects of the ML platform, the inference engine, and some higher prover specific layers. We report on an implementation of all that for Isabelle/Isar, and point out some general considerations for parallelism in functional programming, and other provers like Coq and HOL.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming; I.2.3 [*Deduction and Theorem Proving*]: Inference engines; G.4 [*Mathematical software*]: Efficiency

General Terms Explicit and implicit parallelism

Keywords Isabelle, theorem proving, mathematical assistants, parallel programming

1. Introduction

1.1 The "multicore problem"

According to "Moore's Law", the density of integrated circuitry grows exponentially, doubling the number of transistors per chip area approximately every 20 months. Despite fundamental limitations predicted by physicists, chip designers have managed to keep up to this promise so far, but this does not mean that the extra functionality translates directly into exponential growth of performance.

In fact, only a few years ago processor manufactures have declared an end to sustained increases of traditional single-threaded computing power, but have introduced CPUs with explicitly visible parallel execution units (or "multiple cores"). This essentially imposes an extra burden on application developers, who need to change their implementations in order to turn hardware figures like "8 cores with 2 hyper-threads each" into actual software speedups. Numerous press articles have appeared recently, e.g. declaring that "the free lunch is over" [17] and new software engineering challenges are emerging from this shift of execution paradigm.

On the other hand, the question of effective use of parallelism in application software is rather old. In the past 30–40 years, plenty of research projects have produced a diversity of possible answers. While in hardware and operating system design reasonable multiprocessor support is already commonplace, viable programming languages are only available in some niches.

This programming language aspect shall be our main concern, in particular the situation for computer mathematics, with special focus on interactive theorem proving in the LCF tradition.

1.2 Mathematical assistants with full proof checking

Until the envisioned idea of fully integrated mathematical assistants becomes more tangible, we can only approximate and extrapolate from what has already been achieved so far. We shall do this from the perspective of interactive theorem proving, as represented by the "LCF-style" family of systems: notably Isabelle [20], Coq [18], and HOL variants [8]. The main traits of these systems (inherited from LCF [3]) are as follows.

Full programmability means that the user has access to a general functional programming language to conduct proofs and build derived tools (specific proof procedures, advanced specification mechanisms etc.). In the original LCF system this "meta language" (ML) is implemented as an interpreter within the underlying LISP system. The proof environment is essentially just a collection of ML modules that implement concepts like *goal*, *tactic* etc. In Isabelle and Coq, there are separate interpretation layers for simplified command languages, notably Isar in Isabelle and Gallina/Ltac in Coq. Nonetheless, the ML layer is still available for power users.

At this level we are faced with the parallelization problem in full generality, because user tools can be arbitrary ML programs.

Explicit proof construction means that the main results are based on definitions and theorems. There is no free-form ML code to "invent" results, but there are explicit statements with explicit proofs, e.g. tactic scripts that "drive" certain LCF-style inference rules of HOL, or internalized computations in the more advanced type-theory of Coq.

Consequences of this rigorous style of theory development are:

1. Processing theories requires significant amounts of runtime resources. So we really need to think about using multicore hardware efficiently.
2. Statements impose a natural top-down specification of results, and proofs merely certify these. So proofs can be deferred and checked independently.

This means that the arbitrary ML code mentioned before is actually embedded into some fixed outline of definition–statement–proof that is amenable to automatic parallelization.

So the plan is to reorganize the operational details of the prover such that the inherent potential for parallel proof checking is exploited. User code merely needs to be "well-behaved" to participate — purely functional ML code usually works best. As we shall see later, we need to extend both the ML platform and the proof engine.

All of this could be done similarly with any member of the LCF family, or even with completely different interactive provers. Subsequently, we explain concrete issues for Isabelle/Isar and its underlying SML implementation Poly/ML, while giving further hints on the bigger picture where appropriate.

1.3 Isar proof document structure

The following example in Isabelle/HOL illustrates the general structure of Isar proof documents.

```
theory C imports A B
begin

inductive path for rel :: α ⇒ α ⇒ bool
where
  base: path rel x x
| step: rel x y ⟹ path rel y z ⟹ path rel x z

theorem example:
  fixes x z :: α
  assumes path rel x z
  shows P x z
using assms
proof induct
  fix x
  show P x x  ⟨proof⟩
next
  fix x y z
  assume rel x y and path rel y z
  moreover
  assume P y z
  ultimately
  show P x z  ⟨proof⟩
qed

end
```

Here we have a hierarchy of roughly of 4 layers, with different characteristics concerning the processing order.

1. Theories. There is a directed acyclic graph of theory files, which determines the outer modular structure of the application. In our example, we see one such node *C* that depends on the merge of other nodes *A* and *B*.

   The theory loader can exploit independent paths of this DAG structure and load some theories in parallel. This requires only small changes to the system, and had been our very first attempt at parallelism in Isabelle (in 2007). If a system is built around separate compilation, one could also use external tools like GNU make with option `-j` for parallel compilation.

   In any case, parallel loading of theories is relatively simple to achieve, but performance gains critically depend on the degree of independence of theories within a project. In typical applications this is relatively low, e.g. 1–3 parallel loads at a time.

2. Definitions and statements within the theory body. Here we assume strictly sequential dependencies and refrain from any attempt at parallelizing specifications that happen to be logically independent. This simplistic view is sufficient, because checking of toplevel specifications is usually fast.

   In the example we have a (derived) **inductive** definition, followed by a **theorem** statement. In order to proceed sequentially, the system needs to perform parsing and syntactic type checking, and somehow fork off the associated proofs (see below). Note that **inductive** involves internal proofs of monotonicity of the recursive predicate specification, and the resulting introduction, elimination, induction rules.

3. Toplevel proofs. These are (implicit or explicit) justifications for the definitions and statements within a theory.

   There are two important observations: (1) checking proofs requires most of the total runtime, and (2) proofs are practically irrelevant in the sense that we merely need to know that certification was finished successfully at some point.[1]

   This is our main potential for a high degree of parallelism. In a big project, there are usually hundreds or thousands independent proof checking jobs.

4. Nested proofs. A properly structured proof in Isabelle/Isar consists of a hierarchical outline, where explicit statements are followed by nested sub-proofs. This means we could apply the same principles as for toplevel proofs recursively, but a more basic scheme is already sufficient for the Isar proof language.

   As a design principle of Isar, processing the proof outline and composing local results works by very simple means and requires little runtime. Potentially costly invocation of automated proof tools only occurs in terminal positions, e.g. **by** *auto* instead of some ⟨*proof*⟩ placeholder above. So we merely need to parallelize these terminal proof steps, while the main part of Isar language interpretation is unchanged.

   Here we have another great potential for parallel proof checking in reserve, which we did not exploit in our implementation so far. Toplevel proofs are sufficient to saturate the relatively small number of CPUs on current multicore systems.

The skeletons resulting from Isabelle/Isar document structure lead to *implicit parallelism* with a fairly good granularity of tasks. This almost ideal situation is specific to formal proof checking, and most of our users will continue to get their free lunch [17].

Beyond implicit system-level parallelism, user tools could also use *explicit parallelism*. Internal proof problems could be deferred by using the very same infrastructure as for structured Isar proof processing. Even more ambitious tools might go beyond parallel proof *checking*, and support parallel proof *search*. The latter is a recurrent topic in automated reasoning, which is beyond the scope of the present paper.

1.4 Overview

The rest of the paper is structured as follows. §2 introduces a value-oriented parallel programming model based on "futures". §3 covers parallelism at the level of the inference kernel. §4 briefly reviews the integration with higher-level Isabelle/Isar concepts. §5 discusses concrete performance figures of the actual implementation of Isabelle2009 using Poly/ML 5.2.1.

[1] In Isabelle (and HOL) proofs are formally irrelevant by design of the core calculus and its implementation as LCF-style inference kernel. Nonetheless, optional proof objects can be maintained and used later by other means. Thus we would get some degree of interdependent proofs, although that relation is usually sparse. In Coq the situation is the opposite: terms and proof terms can depend on each other in the formal calculus, but applications often ignore this possibility.

2. Parallel programming model

2.1 Low-level concepts

2.1.1 Hardware

We assume the typical multicore hardware that is now commonplace: a shared-memory multiprocessor with a relatively small number of CPUs (e.g. 2–8) and with a pool of memory that is accessible without any special penalties. We abstract over slight variations seen in reality, such as virtual cores via additional *hyper-threads* on actual CPUs, or memory being accessed in a *non-uniform architecture* where some regions are closer to some cores.

2.1.2 Operating system

We take the well-known *Posix Threads* (or *pthreads*) library for granted. All major operating systems implement a version of this IEEE standard, usually founded in the kernel, such that the raw parallel computing power becomes accessible with very little overhead. Of course, multithreading also works on single-core machines. The scheduler of the operating system will map any reasonable number of threads (e.g. 1–100) to the given number of CPU cores according to some *fair* strategy.

2.1.3 ML runtime system

Our platform of choice is Poly/ML 5.2.1. Thanks to special efforts by David Matthews, it provides an ML view on pthreads (with a few extra abstractions, but little overhead). This ML thread model is the most basic programming interface that we consider here explicitly. There are three main concepts, which are represented as abstract types in ML.

1. Thread: an independent execution of some ML code, with private stack and regular access to the global heap. There is also support for thread-local user data of arbitrary type. The main operations are:

   val *fork*: $(unit \to unit) \to thread$
   val *interrupt*: $thread \to unit$

   This means *fork* (**fn** () ⇒ *body*) will create a new thread that executes *body*, until evaluation terminates regularly or by raising an exception (which is absorbed). Note that there is no *join* operation. In order to simulate a return value, one needs to use side-effects together with suitable synchronization.

   Interrupting a thread from the outside would eventually raise exception *Interrupt* (according to SML90). The exact behavior depends on the state of certain thread attributes, which can be adapted to implement interrupt handlers reliably in user code.

2. Mutex: a passive synchronization primitive. The main operations are:

   val *mutex*: $unit \to mutex$
   val *lock*: $mutex \to unit$
   val *unlock*: $mutex \to unit$

   These mutual exclusion primitives work with little extra overhead. In particular, the common case that locks can be acquired without contention is very fast.[2]

3. Condition variable: an active synchronization primitive, which allows threads to wait for a state change of shared variables, as signalled by another thread. The main operations are:

   val *condvar*: $unit \to condvar$
   val *wait*: $condvar \times mutex \to unit$
   val *signal*: $condvar \to unit$

   The use of a raw pair of $condvar \times mutex$ to work together in the communication and synchronization protocol is typical for pthreads. Textbooks on the subject explain peculiar programming patterns that make this work reliably, to avoid deadlock or starvation (e.g. due to signals emitted in the wrong moment).

This provides a basic ML view on the original C version of pthreads. It still needs to be wrapped into higher-order concepts that are more native to ML.

2.2 High-level concepts

According to Brinch Hansen "concurrent programs can be written exclusively in high-level languages" [2]. Although this advice was targeting Concurrent Pascal, we shall now apply it to ML.

2.2.1 Synchronized variables

First of all we need to represent the idea of controlled access to shared variables adequately. We follow roughly the notion of "conditional critical region" or simple variants of the "monitor" concept due to Hoare and Brinch Hansen. Since the primitives of pthreads are strongly influenced by these mechanisms in the first place, we can easily wrap them into high-level ML operators to reflect the original idea more directly. Our implementation has the following signature:

type α *var*
val *var*: $\alpha \to \alpha\ var$
val *value*: $\alpha\ var \to \alpha$
val *change*: $\alpha\ var \to (\alpha \to \alpha) \to unit$
val *guarded-access*: $\alpha\ var \to (\alpha \to (\beta \times \alpha)\ option) \to \beta$

A synchronized variable v with initial value x is created by invoking *var x*; internally it contains a reference cell with *mutex* and *condvar* for the lock/wait/signal/unlock protocol of pthreads. We can operate on the content via *guarded-access v f*, where the pure function f tells how to proceed: mapping x to *NONE* means we continue to wait for a change of the content (there is a loop internally), and mapping x to *SOME* (y, x') means we update the content to x' and produce a return value y; then all threads waiting on this variable are signalled and the critical region is left.

This rather general guarded access principle can be also used to implement the simpler operations *value* and *change*. Note that after being admitted into the critical region these will never wait.

fun *value v* = *guarded-access v* (**fn** $x \Rightarrow$ *SOME* (x, x))
fun *change v f* = *guarded-access v* (**fn** $x \Rightarrow$ *SOME* $((), f\,x)$)

The subsequent example implements type *MVar* of Concurrent Haskell[3], which represents a buffer that is restricted to 0 or 1 elements. Thus access operations may block in certain cases.

type α *mvar* = α *option var*
fun *mvar* () = *var NONE*
fun *take v* = *guarded-access v*
  (**fn** *NONE* ⇒ *NONE* | *SOME x* ⇒ *SOME* (*x*, *NONE*))
fun *put v x* = *guarded-access v*
  (**fn** *SOME y* ⇒ *NONE* | *NONE* ⇒ *SOME* ((), *SOME x*))

The next example demonstrates a mailbox with unbounded message queue: operation *send* is non-blocking, but *receive* blocks until the mailbox becomes non-empty. Here we assume **type** α *queue* with purely functional operations *empty*, *enqueue* and *dequeue* (as available in the Isabelle/ML library).

type α *mailbox* = α *queue var*
fun *mailbox* () = *var empty*
fun *send mbox msg* = *change mbox* (*enqueue msg*)
fun *receive mbox* = *guarded-access mbox* (*try dequeue*)

[2] Poly/ML uses cheap spinlocks first, and falls back on a more elaborate locking protocol only for longer wait states.

[3] `http://haskell.org/ghc/docs/latest/html/libraries/base/Control-Concurrent-MVar.html`

From the perspective of lower-level languages (like Java), this might look a bit like "domain specific language" or even "aspect oriented programming", but it is really just a regular Isabelle/ML library that has been arranged carefully.

2.2.2 Future values

So far we have managed to provide succinct notation for accessing synchronized variables, but the inherent complexities of concurrent programming are not yet overcome. At the next stage we shall avoid explicitly exposed threads of execution as well, but there are many conceivable ways to do that. For example, there is ongoing research in the Haskell community to eliminate threads and synchronization in favor of *software transactional memory* (STM), which allows *modular composition* of interacting components [7].

Our main purpose is parallel proof checking, and it turns out that the more modest model of *future values* (or *value-oriented threads*) fits to our requirements. Futures are almost folklore in concurrent functional programming. An early implementation is available in Multilisp [5], although that work also points back to previous ideas of *eventual values* in Algol. The Alice ML dialect [15] also includes futures as one of many builtin concurrency concepts.

Futures in Isabelle/ML implement the following signature:

type α *future*
val *future*: $(unit \to \alpha) \to \alpha\ future$
val *join*: $\alpha\ future \to \alpha$
val *cancel*: $\alpha\ future \to unit$

The idea is that *future* (**fn** () $\Rightarrow$ *expr*) produces a handle to the eventual result of evaluating *expr*. The actual computation will commence spontaneously in the background at some point in the future. The *join* operation re-synchronizes and exhibits the result — this could mean to wait until it is produced by another thread, or to force its evaluation at that very moment. As usual in ML, the result of an evaluation may either produce a plain value, or raise an exception; the latter is propagated to the synchronization point of *join*. For example, **val** $x =$ *future* (**fn** () $\Rightarrow$ *raise Match*) succeeds, but any later *join x* will raise exception *Match*.

The *cancel* operation provides some minimal control over the evaluation process: canceling an unfinished future means to force it into *Interrupt* exception state, but finished values are unchanged.

Futures provide a specific model of *managed evaluation* under program control. For example, we can implement the list combinator *parallel-map*: $(\alpha \to \beta) \to \alpha\ list \to \beta\ list$ as follows:

fun *parallel-map f xs* =
  map join (*map* (**fn** $x \Rightarrow$ *future* (**fn** () $\Rightarrow f\ x$)) *xs*)

By using combinators like this, the user can already build parallel programs via *skeletons* that indicate the boundaries of parallelism explicitly. Although this style is much more convenient than working with raw threads, it is still quite hard to find just the right spots to insert parallel combinators, in order to achieve actual performance improvements. It is desirable to support *implicit parallelism* somehow, where the system (or compiler) determines the granularity of parallelism, but this is very hard in full generality (e.g. see [16]).

As already outlined in §1.3, we are very fortunate in this respect: the proof document structure already provides quite good skeletons for implicit parallelism.

Implementing futures. Two main aspects need to be addressed: providing a slot for the result and maintaining an unevaluated expression that produces the result eventually. In fact, this basic pattern is the same as for lazy evaluation in strict ML. For futures, the key difference is that a separate *scheduler* in the background ensures that evaluation is invoked eventually; the actual execution is then performed by a farm of *worker threads*.

Our future values are represented as follows:

datatype α *result* = *Result of* α | *Exn of exn*
type α *future* = α *result option ref* $\times$ *task*

The result field is initialized as *ref NONE* for a newly created future. The associated *task* is a symbolic representation of an actual evaluation process that is associated with a *job*: $unit \to unit$ that runs the evaluation and sets the result to *SOME result* eventually.[4]

There are 3 phases in the lifetime of a future task:

1. *queued:* the task is stored in the priority queue of the scheduler, waiting passively to be picked up by some worker thread.
2. *running:* some worker thread has assumed the role of executing the job that is associated with the task. This state is indicated both in the thread-local data of the worker and in the queue.
3. *finished:* the worker has finished evaluation; the task is removed from the queue altogether.

The main purpose of the scheduler is to feed tasks of newly created futures into the queue, and ensure that a given number of worker threads is running (this parameter can be adjusted at runtime). The workers wait on changes of the task queue, and pick the next job that is ready to be executed: tasks can depend on each other, only minimal elements of the queue can be selected here.

Invocation of *join x* for unfinished *x* requires special attention. If the current thread is a worker running another task, it will take over execution of the next task that contributes to *x* according to queue dependencies, or any other task that happens to be ready. Otherwise, if this is a non-worker thread, *join x* waits until the future is finished independently. Workers only ever sleep if there are no ready tasks in the queue.

This simple scheduling policy ensures that the available processor resources are saturated as much as possible, with a number of worker threads that corresponds to the number of CPUs.

Another key observation concerns concurrent access to the result field of a future:

Read access can happen at any time (via arbitrary invocations of *join*), but it requires a finished result of the form *SOME result*.

Write access happens at most once (via some worker thread), by flipping the initial *NONE* to a persistent value *SOME result*.

This *monotonicity principle* justifies the terminology of "future value" in the first place. It also means that careful implementation of access primitives can work without extra synchronization: there is no overhead to read the value of a finished future.

Synchronization is only required when accessing internal state variables of the scheduler, notably the global task queue. In fact, queue management turned out a real bottleneck in an earlier version and required some further fine tuning. The first idea was to improve the basic data structure implementation. The second, even more important idea was to provide fast-path implementations of the following operations:

fun *future-value a* = *future* (**fn** () $\Rightarrow a$)
fun *future-map f x* = *future* (**fn** () $\Rightarrow f$ (*join x*))

These operations are often required to adapt to an interface that expects a future, or produce variants of futures via (quick) projections. The latter essentially reflects conceptual layers of the overall system, for example *Proof.state future* is mapped to *thm future*, which is mapped to *Proofterm.proof future* eventually (cf. §4). By providing a direct way to extend the task of an unfinished future on the fly, we avoid an extra penalty for modular software organization.

[4] Interestingly, the scheduler only needs to manage a homogeneous collection of tasks/jobs, each operating on a statically typed reference within its functional closure. This is the deeper reason why typed futures can be implemented as plain ML library, without requiring a universal type of types.

3. Parallel inference kernel

The main LCF-style inference kernel of Isabelle/Isar consists of two layers: proof objects and theorems.

Proof objects are represented as concrete datatypes. Depending on system parameters, a varying amount of information is maintained here. In any case, there is a complete record of oracles used in the proof. An explicit λ-term representation of the proof is also possible, but requires significantly more resources.

Theorems are elements of an abstract datatype *thm*. According to the original LCF tradition, theorems are proven propositions that have been certified by the kernel module, which implements primitive inferences as abstract datatype constructors. Any operation on theorems has to go through that kernel, so any value of type *thm* is "correct by construction", cf. [3]. Isabelle theorems consist of the the proof object, the proven proposition, plus some extra administrative information. Most notably, there is an explicit certificate of the background theory of the theorem.

In order to support parallel proof checking, we need to augment both layers. The idea is to allow proof objects to contain holes that essentially act like axioms that can be replaced later. This basic form of *promised proofs* that are fulfilled eventually is then passed through the abstract type *thm*. Thus we extended the LCF inference kernel to support parallelism natively, without affecting final results: there is no trace of the operational details of proof composition in a finished derivation.

3.1 Promised proofs

The core inference system of Isabelle is based on Natural Deduction, deriving judgments of the form $\{A_1, \ldots, A_n\} \vdash B$ with rules for $\Longrightarrow$ and $\bigwedge$ that move propositions between the antecedent Γ and the succedent B. Subsequently, we include explicit proof objects as λ-terms, although they might be omitted in the implementation. So $p : B$ means that p is a proof term for proposition B.

Inferences are always relative to a background theory Θ, which contains constants for types, terms, and proofs (axioms), as well as any kind of auxiliary user data [21]. Isabelle maintains certificates for Θ, such that $\Theta_1 \subseteq \Theta_2$ and $\Theta_1 \cup \Theta_2$ can be determined efficiently. Below it is sufficient to think of Θ as a set of axioms.

Proof holes work like global axioms, but need to be managed separately so that they can be fulfilled later (by proof substitution). For the moment we introduce a pro-forma collection Π of promised proofs, and pass it through monotonically. Thus we can describe the deductive system of Isabelle as a judgement $\Theta, \Pi, \Gamma \vdash p\colon B$ that is defined inductively by the following rules:

$$\frac{\Theta, \Pi, \Gamma \vdash q : B}{\Theta, \Pi, \Gamma - \{p : A\} \vdash (\lambda p : A.\ q) : (A \Longrightarrow B)} \quad (\textit{imp-intro})$$

$$\frac{\Theta_1, \Pi_1, \Gamma_1 \vdash p : (A \Longrightarrow B) \quad \Theta_2, \Pi_2, \Gamma_2 \vdash q : A}{\Theta_1 \cup \Theta_2, \Pi_1 \cup \Pi_2, \Gamma_1 \cup \Gamma_2 \vdash p\ q : B} \quad (\textit{imp-elim})$$

$$\frac{\Theta, \Pi, \Gamma \vdash p[x] : B[x] \quad x \notin \mathrm{FV}\,\Gamma}{\Theta, \Pi, \Gamma \vdash (\lambda x.\ p[x]) : (\bigwedge x.\ B[x])} \quad (\textit{all-intro})$$

$$\frac{\Theta, \Pi, \Gamma \vdash p : (\bigwedge x.\ B[x])}{\Theta, \Pi, \Gamma \vdash p\ a : B[a]} \quad (\textit{all-elim})$$

$$\frac{}{\Theta, \Pi, \{p : A\} \vdash p : A} \quad (\textit{assm})$$

$$\frac{(c : A[?\overline{\alpha}]) \in \Theta}{\Theta, \emptyset, \emptyset \vdash c : A[\overline{\tau}]} \quad (\textit{axiom})$$

$$\frac{\Theta, \Pi, \Gamma \vdash p[\alpha]\colon B[\alpha] \quad \alpha \notin \mathrm{TV}\,\Gamma}{\Theta, \Pi, \Gamma \vdash p[?\alpha] : B[?\alpha]} \quad (\textit{type-gen})$$

$$\frac{\Theta, \Pi, \Gamma \vdash p[?\alpha] : B[?\alpha]}{\Theta, \Pi, \Gamma \vdash p[\tau] : B[\tau]} \quad (\textit{type-inst})$$

Taking $\Pi = \emptyset$ for now, we get the existing inference system of Isabelle/Pure [19], which we shall use as a reference point for correctness of the extended version introduced below.

Rules *type-gen* and *type-inst* require special attention. This is how Isabelle (and the HOL family in general) simulate outermost type quantification in the style of ML polymorphism [11]. In Isabelle notation, α refers to a *free* and $?\alpha$ to a *schematic* type variable; this provides a syntactic hint if a type is considered locally fixed or arbitrary. The above type manipulations essentially emerge as admissible rules (by induction over derivations) as follows: the base case works because the axiomatic basis of Θ is closed by arbitrary type substitutions; the step case works by pushing substitutions through the other inferences.

This form of naive polymorphism introduces an extra twist in maintaining promised proofs, because we need to track type substitutions and replay them when proofs are fulfilled eventually. If we would not care about schematic polymorphism, deferred proofs could be simulated outside the kernel as assumptions [1].

A *proof promise* is a constant of the form $a[?\overline{\alpha}]\colon A[?\overline{\alpha}]$ where a is some identifier and A a closed proposition that contains only schematic type variables $?\overline{\alpha} = ?\alpha_1, \ldots, ?\alpha_n$ (in canonical order). Promises shall be treated like axioms with an explicit indication of type instances; rules *type-gen* or *type-inst* will produce concrete instances $a[\overline{\tau}]$ at different occurrences in a proof term. Moreover, we define proof substitution $p[a := q]$ for a closed proof $q : A$ (we assume $\mathrm{TV}\,q = ?\overline{\alpha}$ without loss of generality):

$$\begin{array}{rcl}
(a[\overline{\tau}] : A[\overline{\tau}])\ [a := q] & = & q[\overline{\tau}] \\
(b[\overline{\tau}] : B[\overline{\tau}])[a := q] & = & (b[\overline{\tau}] : B[\overline{\tau}]) \quad \text{if } a \neq b \\
(c : A)[a := q] & = & (c : A) \\
(p_1\ p_2)[a := q] & = & (p_1[a := q])\ (p_2[a := q]) \\
(\lambda x.\ p)[a := q] & = & \lambda x.\ p[a := q]
\end{array}$$

In other words, we treat q as a definition for a that is expanded throughout the proof term, while propagating type substitutions accordingly. This is analogous to polymorphic *let* expressions [11]. It is particularly important to note here that substituting a closed term does not interfere with λ-abstraction.

To ensure well-defined substitution of proof promises by proofs depending on further promises, we postulate some well-founded relation $a_1 < a_2$ on promise identifiers. For example, we can identify promises via natural numbers with the standard ordering; whenever a proof promise is opened we produce a fresh (monotonically increasing) serial number.[5] The order is lifted to a set of promises as follows: $\Pi \ll a$ iff $b < a$ for all $(b[\overline{\tau}] : B) \in \Pi$.

The parallel version of the Isabelle inference system is now defined inductively by the following additional rules:

$$\frac{\mathrm{FV}\,A = \emptyset \quad \mathrm{TV}\,A = \{?\overline{\alpha}\}}{\Theta, \{a : A\}, \emptyset \vdash a[?\overline{\alpha}] : A[?\overline{\alpha}]} \quad (\textit{promise})$$

$$\frac{\Theta_1, \Pi_1, \Gamma \vdash p : B \quad \Theta_2, \Pi_2, \emptyset \vdash q : A \quad \Theta_2 \subseteq \Theta_1 \quad \Pi_2 \ll a}{\Theta_1, (\Pi_1 - \{a : A\}) \cup \Pi_2, \Gamma \vdash p[a := q] : B} \quad (\textit{fulfill})$$

[5] This well-founded order is sufficient for bottom-up proof parallelization, corresponding to holes in a proof skeleton that emerge by depth-first traversal. One could also think of more general situations, e.g. where the main proof is first forked in top-down fashion and its sub-proofs are then assembled bottom-up. This would require a more flexible ordering on promises.

In fact, the resulting inference system merely formalizes the idea of introducing and eliminating locally defined proof terms, analogous to polymorphic *let* expressions as mentioned before. Parallelism is only implicitly present: *promise* produces a slot for a future certification of $a : A$, and *fulfill* joins a derivation of $p : B$ that depends on $a : A$ with an actual $q : A$ produced independently. Note that the requirement $\Theta_2 \subseteq \Theta_1$ ensures that the resulting theory is unchanged: $\Theta_1 \cup \Theta_2 = \Theta_1$.

3.2 Future theorems

In order to integrate proof terms with promises (§3.1) into the LCF-style inference kernel, the internal representation of type *thm* is augmented by an environment of pairs (a, q) where a is a promise identifier and q a future proof. Whenever a promise $a : A$ is created, its eventual fulfillment has to be given as well, as a future $q : A$ that can be produced in parallel (§2.2.2). Thus we get the following additional operations of the inference kernel:

```
val future-thm: thm future → term → thm
val join-proof: thm → unit
```

For example, *future-thm* (*future* (**fn** () ⇒ *prf*)) *A* produces a theorem of proposition *A*, but the actual proof is delivered later by evaluating the given *prf*: *thm*. The kernel ensures that the result really fits the original specification *A*, and checks the side-conditions of the *fulfill* inference (§3.1), notably background theory inclusion and well-founded dependencies on other promises.

Observe how the signature of *future-thm* formally reflects our main observation of "practical proof irrelevance" (§1.3): a given *thm future* together with an explicit result specification as a *term* is turned into a plain *thm*. So proof parallelization is isolated locally: it will not affect the great majority of existing ML code. Internal bookeeping of promised proofs is fully transparent: when requesting the explicit proof object, the kernel will *join* and *fulfill* all pending proof promises (this can take a long time). Any runtime failures of forked derivations etc. will be raised at that point.

As a minor drawback we loose purely static integrity of LCF kernel results: entities of type *thm* may contain holes that might fail to be filled, because some future proofs did not work out as promised. Explicit dynamic checking via *join-proof* is required here, to ensure that all open ends are really closed.

This notable change of the LCF kernel semantics has very little impact in practice. In Isabelle/Isar theorems that are accessible to the user are always registered within the theory context eventually, so we merely need to ensure that *join-proof* is invoked on that cumulative collection at certain checkpoints of the theory loading process. Thus full integrity is recovered at the outer theory level.

4. Isabelle/Isar integration

With the basic LCF kernel extensions of §3.2 available, we can now go throw the higher layers of Isabelle/Isar and handle future proofs accordingly.

4.1 Open statements

Due to side conditions in the *promise* inference (§3.1), *future-thm* (§3.2) is still limited to closed statements.

Open statements (relative to some local proof context) routinely occur in practice, even at the top level. For example, the theorem in §1.3 builds up a local context via **fixes** and **assumes** and then establishes a local result via **shows**. Note that types are always fixed implicitly according to Hindley-Milner discipline [11].

We can easily produce an enhanced version of *future-thm* by commuting the future value with abstraction and application of local types, terms, and assumptions like this:

```
fun local-future-thm prf $\overline{\alpha}$ $\overline{x}$ $\overline{A}$ B =
  let
    val global-prf = future-map (abstract-proof $\overline{\alpha}$ $\overline{x}$ $\overline{A}$) prf
    val global-prop = abstract-term $\overline{\alpha}$ $\overline{x}$ $\overline{A}$ B
  in apply-proof (future-thm global-prf global-prop) $\overline{\alpha}$ $\overline{x}$ $\overline{A}$ end
```

Note that "abstraction" over types is simulated via schematic variables, and application is type instantiation. For term parameters and assumptions, the introductions and eliminations of $\bigwedge$ and $\Longrightarrow$ are used. The result will be the same as for an immediate proof — thanks to β-equivalence on primitive proof terms — but this imposes a small operational overhead.

4.2 Goal-directed proofs

As has already been observed in the original LCF system [3], user-level proof programming mostly proceeds in a goal-oriented style, via tactics. A *tactic* is essentially a function that reduces some claim to a number of new claims. The goal mechanism produces an initial state, applies a given tactic expression, and checks that all claims are solved in the end.

In Isabelle/Isar this idea is generalized a bit to work with local contexts. The main programming interface is essentially as follows:

```
val Goal.prove: Proof.context → term → tactic → thm
```

The implementation can be easily refined to make use of future theorems for the main result, thus any such goal-directed proof would be parallelized. On the other hand, existing applications occasionally expect strict behavior of the goal interface, where failure of the tactic needs to be reported immediately.

So we keep *Goal.prove* unchanged, but provide a second version *Goal.prove-future* (with the same signature). User code can be fine-tuned by replacing *Goal.prove* by *Goal.prove-future* wherever appropriate. For example, in **inductive** definitions (cf. the example in §1.3), the monotonicity proof is always performed immediately to report fundamental errors in the specification, but all other derived rules are produced via future goals.

4.3 Structured Isar proofs

This refers to the actual Isar proof interpretation process. Conceptually, the Isar language is very modular, so that arbitrary sub-proofs could be deferred. We only do this for the main toplevel proof for now, because that turns out as sufficient to saturate 4–8 cores reasonably well, and due to the simplistic well-founded ordering on promises based on serial numbers (§3.1).

Nonetheless the operational details are a bit tricky, because Isar proof interpretation works in a sequential step-by-step basis, due to its heritage of the TTY-based LCF interaction model. We essentially need to modify theory source processing to group statements with proofs by static look-ahead, which requires some care since the language is fully extensible by the user.

In the end, the refined system supports fully implicit Isar proof parallelization, without any impact on user code. This is where our concept of parallel proof checking is realized in its purest form.

4.4 Theory loading

The Isabelle theory loader processes a DAG of theory files, essentially by turning every single load process into a future, while recording dependencies properly. Despite the relative simplicity of theory loading, some fine points need to be observed.

- Joining proofs. All pending future proofs need to be joined, to exhibit potential errors, cf. *join-proof* in §3.2.

  The speedup factor is maximized by doing a global join after *all* load processes have been forked, such that the highest possible degree of parallelism is achieved. On the other hand, this requires substantial amounts of memory, due to a large number of intermediate proof configurations.[6]

- Document preparation. The main result of Isabelle/Isar theory processing are formally checked LaTeX documents. This used to work as a "side-effect" of theory loading, but requires special attention in parallel mode. Firstly, old stateful document accumulation needs to be done in a more disciplined manner. Secondly, printing of formal entities (via document antiquotation) requires extra care, to avoid premature joining of proofs — otherwise performance will degrade.

In summary, there are two main lessons learned from pushing parallel proof checking through all layers of Isabelle/Isar.

1. System infrastructure needs significant refinements: there is no magical way to add the "aspect of parallelism" automatically.
2. User code requires only minimal fine-tuning occasionally.

5. Performance

It is time for some quantitative assessment of our infrastructure for parallel proof checking. *Amdahl's Law* is a simple model that predicts limits of the overall speedup factor as follows. Given a program whose runtime is partitioned into a fraction s that is inherently sequential and the complement p that can be parallelized. Then standardized runtime is $1 = s + p$ in the sequential case, and $s + p/n$ in the parallel case for n cores. So the speedup factor is $1\ /\ (s + p/n)$, which converges to $1/s$ for $n \longrightarrow \infty$.

For example, if 20% of the program is sequential, the maximum speedup is only 5, even if we have infinitely many cores! This estimate is indeed quite pessimistic, and there are situations where better performance can be achieved — even super-linear speedup if disjunctive branches of proof search would be tried in parallel, for example. On the other hand, we check existing proofs that happen to be independent anyway, so we need to be prepared for more conservative results.

For our concrete measurements presented below, we use a first generation MacPro with 4 cores (2 dual-core Xeons), Poly/ML 5.2.1 with 2 GB initial heap, and Isabelle2009 with parallel loading of theory files and parallel checking of proofs both enabled. The test sessions are HOL-Auth (with many independent theories), HOL-Nominal (most of the time spent in a single large theory), and HOL (the main logic with many sequential bottlenecks).

For each session we measure the elapsed "wall-clock time" (which is in the range of $2-15$ minutes), and the CPU time spent cumulatively in the lifetime of the process (as reported by the operating system). We do this for $n = 1..4$, where the sequential case is measured in two ways: 1 worker thread using the parallel infrastructure vs. a truely sequential variant that bypasses all this extra overhead.

Figure 1 shows the *relative speedup*, i.e. the ratio of elapsed time vs. CPU time for each given session.

As is typical for sub-linear speedups according to Amdahl's Law, the curves are flattening with increasing number of cores. Both HOL-Auth and HOL-Nominal show quite good speedup factors of 3.2 for 4 cores: there are many independent theories or independent proofs to be run in parallel, and it seems we are able to exploit this reasonably well. The HOL session is much worse: only rarely does it admit significant parallel tasks, as can be checked

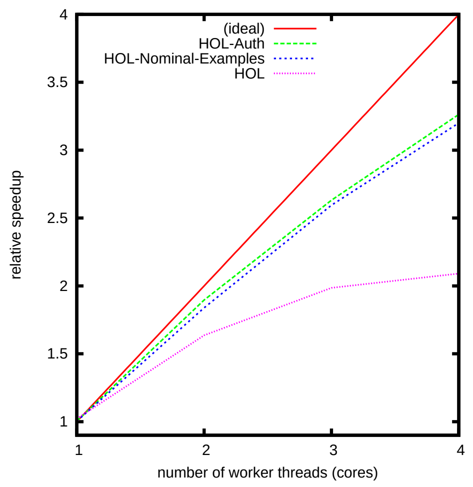

Figure 1. Relative speedup

with the Unix `top` utility. This particularly pathologic situation is special to the initial HOL session, because it contains many inherently sequential stages of bootstrapping sophisticated tools, and relatively few longer proofs. Moreover, if we enable full proof terms for HOL, we run into a real problem: overall memory usage of deferred proof states together with non-normalized proof objects exceeds 32 bit address space, and we need to fall back on theory-only parallelism. Thus the speedup declines to less than 1.3 for 4 cores.

Compared to other Isabelle applications, HOL-Auth and HOL-Nominal represent the best case, and HOL the worst case.

The chart in figure 2 is a bit more realistic in presenting the *absolute speedup*, i.e. the ratio of elapsed time of multicore vs. strictly sequential execution.

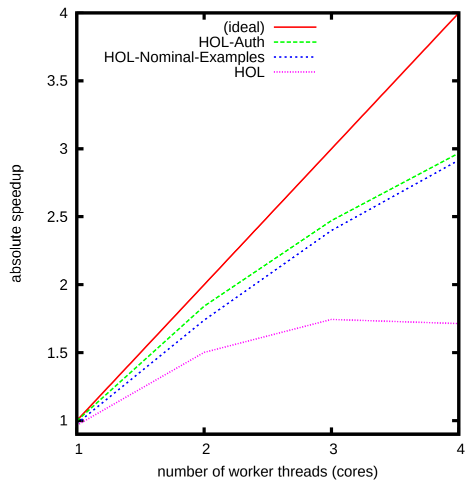

Figure 2. Absolute speedup

Gladly, the difference is not that big: real-world speedup is still 3.0 for 4 cores in the best case, although the worst case already shows a slight decline when moving from 3 to 4 cores. This is due to extra internal overhead for managing parallelism in the first place. To quantify these losses, we show the ratio of cumulative CPU time in multicore vs. the strictly sequential execution in figure 3.

This relative overhead accounts for task management and tracking of proof promises within our derivation objects (every single in-

[6] Normally, a highly parallel system also provides plenty of memory, but we have encountered situations (only with full proof terms) where the 32 bit address space was exhausted. Using Poly/ML in 64 bit mode almost doubles the base-line requirements for CPU time, because every single value is represented as one full machine word.

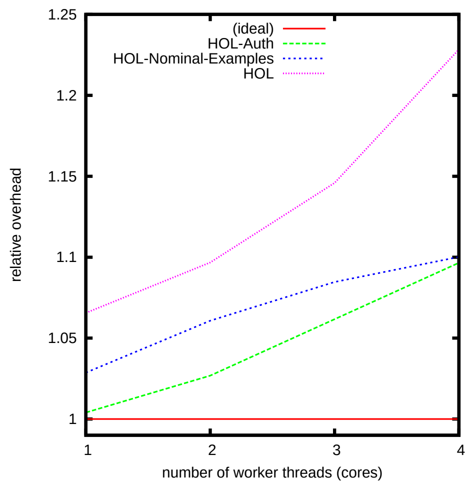

Figure 3. Relative overhead

ference needs to take account of open “holes” in the proof). Again, the numbers are quite good: only $3-10\%$ overhead in the best case, and 22% overhead in the worst case.

Compared with speedup charts for completely different applications in the literature (e.g. [9, 6, 10]) our results are indeed quite competitive. Nonetheless, the curves will inevitable flatten when adding more cores: preliminary experiments on other people’s 8-core machines indicate a speedup of ≈ 5 for 8 worker threads.

There are many fine points that can prevent good performance figures for parallel computation. In fact, in our first attempts the speedup was only $1.5-2.0$ for 4 cores. Practical performance tuning affects the following parts of the overall system:

- Fine tuning of the future scheduler, notably the global task queue, which involves an explicit dependency graph.
- Direct support for projected futures (cf. *future-map* in §2.2.2), which reduces the number of managed tasks significantly.
- Isolating critical points in the system that need to be parallelized separately.[7]
- Eliminating global critical sections of user code by disposing impure programming techniques with global references.

Some further sequentialization is imposed by the underlying Poly/ML runtime system due to garbage collection. The GC thread stops all user threads until it is finished! In realistic Isabelle sessions, GC time is about $5--15\%$, which adds to the critical parameter s in Amdahl’s Law. We can reduce GC time slightly by increasing initial heap size to use most of the available memory from the very start: deferred GC will find fewer live objects to take care of. Additional small-scale tuning can be done by avoiding excessive heap allocations of old tail-recursive “optimizations”, and working with plain recursion instead. Thus we trade global heap allocations for relatively cheap thread-local stack space.

Of course, one could consider to refine the Poly/ML runtime system to support truely parallel GC, which can mean GC in parallel to the user threads, or GC that stops user threads but runs in parallel internally. Both can be done, but is very complex. Very few widely available runtime systems support truely parallel GC, with the notable exception of the Hotspot server JVM by Sun. Even Glasgow Haskell used to support only sequential GC, until the most recent 6.10 release where some promising experiments [10] for parallelized stop-the-world GC have been included.

Our experience of several months of implementation efforts and fine-tuning of parallel Isabelle can be summarized as follows:

- Purely functional code and data leads to correct functionality relatively easily, and is also fast because it does not inhibit parallelism by unnecessary synchronization.
- Imperative techniques require extreme care, much more than in the sequential case. Overall performance is usually degraded due to forced sequentialization.

In other words, impure programming might well be considered as premature optimization from the past that is better avoided in highly parallel programs — if correctness and performance matter.

6. Conclusion and related work

We have presented a reasonably simple and efficient parallel programming environment that is specifically targeted at LCF-style theorem proving. Our particular implementation for Isabelle/Isar allows users to benefit from mainstream hardware immediately, without having to worry about parallelization themselves. In other words, Isabelle acts like system software here: it manages checking of regular proof texts, which can appeal to user-defined proof tools and derived specification packages.

All of the concepts described here are already implemented in the current Isabelle2009 release and work with Poly/ML 5.2.1 on stock multicore systems (both Linux and Mac OS have been tested extensively). Despite fundamental changes in the proof checking architecture, everything worked out quite well and was finished within a few person months. The deeper reasons for this successful shift of the very execution paradigm are as follows.

- LCF-style theorem proving fits nicely to the idea that proofs are delivered later, produced by independent threads of execution — proofs are never inspected in the original LCF approach anyway. This model works particularly well in goal directed proof, because results are fully specified in advance.
- The existing body of Isabelle/ML sources was already *almost* purely functional. We merely had to throw out a small amount of stateful code that had crept in over the years.
- Isabelle did already provide powerful abstractions of formal reasoning environments (ML type *theory* and *Proof.context*) that were easily extended and re-interpreted as thread-local execution contexts of independent proofs.

In principle, other members of the LCF family could be adapted in a similar way, although there are some additional issues.

In Coq [18] proof terms are recorded explicitly by default, which means one needs to care more about efficient management of “proof holes” and substitution of the same. We have already hinted at substantial performance losses with main Isabelle/HOL and full proof terms, which we did not investigate to the last consequence, because proof terms are rarely used in Isabelle at all. Moreover, the more complex type-theory of Coq (and similar dependently-typed calculi) involves formal dependencies between proofs. This reduces the potential for proof parallelism, although the practical impact probably depends on the actual application.

The HOL [8] family is much closer to Isabelle in many respects, but implementing parallel proofs is probably harder due to the lack of higher Isar infrastructure. In particular, the traditional HOL variants operate implicitly on one big theory context that is extended monotonically. Thus it is technically more difficult to fork side branches of independent proof checking and join them back later.

[7] In Isabelle the begin and end of a theory involves certain bookkeeping that can be slow, and requires extra attention here.

Parallelism in functional programming languages. Functional programming has been proposed as an ideal paradigm for parallelism for many decades already. There have been numerous research prototypes, but very few systems ever made it into realistic implementations that are used widely.

Although LCF-style interactive theorem proving fits particularly well with parallelism, the traditional ML platforms for these systems mostly fail to deliver it. Although there is support for threads and other basic concurrency primitives in OCaml, Mlton, SML/NJ, and especially in Alice, none of the underlying runtime systems work with truely parallel system threads. For SML/NJ there are some ongoing experiments to make the old Concurrent ML subsystem work with multiple cores [13], but preliminary performance figures are still relatively low [14].

Poly/ML 5.2.1 is exceptional in the ML family, with high-performance access to native pthreads, although there is only sequential garbage collection so far.

In the larger functional community, Glasgow Haskell appears to be the main platform that supports a wealth of concurrency concepts and achieves good multicore performance, e.g. see [6, 10]. There are many papers and course notes on concurrent and parallel Haskell, and there is active research in various directions to improve this even more, e.g. [7].

Scala [12] is another emerging player in the league of high-end languages; it targets the JVM by default. Conceptually, Scala is many steps beyond Java: it supports pure values (which are called *immutable objects*), higher-order functions, advanced functional and object-oriented composition principles etc. By careful design it also shares representations of basic objects and classes with Java, which allows to reuse existing libraries directly. Scala inherits the usual (dis)advantages of the JVM, which means that raw execution speed is lower than Haskell or Poly/ML, but there is well-established support for true parallelism (including GC).

Due to some unorthodox syntax conventions, Scala supports the idea of "domain specific languages" even better than ML or Haskell. There is a nice Scala version of the concurrent *actor* concept from Erlang, implemented as a library in the Scala distribution. There is ongoing research on rephrasing old concurrency problems using actors and other higher-order abstractions [4].

It would be interesting to see how LCF-style provers would work either on Glasgow Haskell or Scala/JVM (e.g. on Sun Niagara with 128 threads). On the other hand, existing theorem provers like Isabelle, Coq, and HOL are huge and complex systems, so porting to a different programming language is infeasible. As we have demonstrated here for Isabelle/Isar, it is indeed much easier to adapt the given programming language and runtime system, even if the main concurrency infrastructure has to be built from scratch. We hope that this will be repeated for similar provers on similar programming language environments eventually.

Acknowledgments

David Matthews managed to implement true system-level multi-threading for Poly/ML within a few months in 2007/2008. In fact, his parallel ML project goes back to the late 1980-ies, when classic CISC platforms like the VAX where reaching architectural limits and going parallel, but subsequent CPU designs (RISC, P5, ...) managed to postpone the "multicore problem" by some 20 years.

References

[1] H. Amjad. Shallow lazy proofs. In J. Hurd and T. F. Melham, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603 of *LNCS*. Springer, 2005.

[2] P. Brinch Hansen. Monitors and concurrent Pascal: a personal history. In *ACM SIGPLAN conference on History of programming languages (HOPL-II)*. ACM, 1993.

[3] M. Gordon, R. Milner, L. Morris, M. C. Newey, and C. P. Wadsworth. A metalanguage for interactive proof in LCF. In *Principles of programming languages (POPL)*, 1978.

[4] P. Haller and M. Odersky. Scala actors: unifying thread-based and event-based programming. *Theoretical Computer Science*, 2008.

[5] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7 (4), 1985.

[6] T. Harris, S. Marlow, and S. Peyton Jones. Haskell on a shared-memory multiprocessor. In *ACM SIGPLAN workshop on Haskell*. ACM Press, September 2005.

[7] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. *Commun. ACM*, 51(8), 2008.

[8] J. Harrison, K. Slind, and R. Artan. HOL. In Wiedijk [22].

[9] H.-W. Loidl et al. Comparing parallel functional languages: Programming and performance. *Higher Order Symbol. Comput.*, 16(3), 2003.

[10] S. Marlow, T. Harris, R. P. James, and S. Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *International Symposium on Memory Management*, 2008.

[11] R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, (17), 1978.

[12] M. Odersky et al. An overview of the Scala programming language. Technical Report IC/2004/64, EPF Lausanne, 2004.

[13] J. Reppy and Y. Xiao. Toward a parallel implementation of Concurrent ML. In *Workshop on Declarative Aspects of Multicore Programming (DAMP)*, January 2008.

[14] J. Reppy, C. Russo, and Y. Xiao. Parallel Concurrent ML. In *ACM SIGPLAN International Conference on Functional Programming (ICFP 2009)*. ACM, September 2009.

[15] A. Rossberg, D. Le Botlan, G. Tack, T. Brunklaus, and G. Smolka. Alice through the looking glass. In *Trends in Functional Programming*, volume 5. Intellect Books, Bristol, UK, 2006.

[16] N. Scaife, S. Horiguchi, G. Michaelson, and P. Bristow. A parallel SML compiler based on algorithmic skeletons. *J. Funct. Program.*, 15(4), 2005.

[17] H. Sutter. The free lunch is over — a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[18] L. Théry, P. Letouzey, and G. Gonthier. Coq. In Wiedijk [22].

[19] M. Wenzel. *The Isabelle/Isar Implementation Manual*. TU Munich, 2009. Part of the Isabelle2009 distribution.

[20] M. Wenzel and L. C. Paulson. Isabelle/Isar. In Wiedijk [22].

[21] M. Wenzel and B. Wolff. Building formal method tools in the Isabelle/Isar framework. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2007)*, volume 4732 of *LNCS*. Springer, 2007.

[22] F. Wiedijk, editor. *The Seventeen Provers of the World*, volume 3600 of *LNAI*. Springer, 2006.