Reasoning between Programming Languages and Architectures

Francesco Zappa Nardelli

École normale supérieure

Mémoire d'habilitation à diriger des recherches

Specialité Informatique

Reasoning between Programming Languages and Architectures

Francesco Zappa Nardelli

Présenté et soutenu publiquement le 16 janvier 2014, après avis de :

Gilles Barthe	*IMDEA Software Institute, Espagne*
Lars Birkedal	*Aarhus University, Denmark*
Hans Boehm	*HP Labs, États-Unis*

devant la commission d'examen formée de :

Gilles Barthe	*IMDEA Software Institute, Espagne*
Lars Birkedal	*Aarhus University, Denmark*
Giuseppe Castagna	*CNRS & Univ. Paris Diderot, France*
Andrew Kennedy	*Microsoft Research, Royaume-Uni*
Xavier Leroy	*INRIA, France*
Xavier Rival	*CNRS & École normale supérieure, France*
André Seznec	*INRIA, France*

Contents

Prologue

What's past is prologue;
what to come in yours and my discharge.

I defended my PhD on proof methods for process languages in December 2003. I then spent one year as post-doc in the Computer Laboratory at the University of Cambridge and I have been employed as *chargé de recherche* in the Moscova project-team shortly thereafter. During my years at INRIA I have enjoyed a priceless freedom that allowed me to work on a wide range of topics. For this I am deeply indebted to Jean-Jacques Lévy, and later to Marc Pouzet, and to the numerous researchers and students I had the luck to meet and the pleasure to work with.

This *mémoire d'habilitation* consciously focuses on four research directions that I am considering for the forthcoming years. To be accessible to a wider public the presentation is intentionally non-technical, details are to be found in the published papers.

For completeness, below I give a brief overview of all my research contributions since my PhD defense in 2003.

Contribution 1.
Interaction between programs: safe marshalling, integration of typed and untyped code

My research on programming languages, in particular on language design and reasoning techniques for concurrent and mobile computation, began with a collaboration with Peter Sewell's group at University of Cambridge. We developed principled support for type-safe interaction between distributed programs: addressing dynamic loading, rebinding to local resources, global naming for types and terms, versioning, code and computation mobility, modularity. These ideas were realised in the Acute prototype language (available from `http://www.cl.cam.ac.uk/~pes20/acute/`), and published in [88, 89].

We also released a complete specification of the Acute programming language: the lack of suitable tools for to complete this task motivated the development of tool support for semantics (presented as Contribution 2).

More recently, in collaboration with Jan Vitek (Purdue University, USA) and Tobias Wrigstad (Uppsala University, Sweden), I focused on language support for integration of typed and untyped code in the same system. For this, we introduced like types, a novel intermediate point between dynamic and static typing: occurrences of like types variables are checked statically within their scope but, as they may be bound to dynamic values, their usage is checked dynamically. This allows aggressive optimisations on the strongly typed parts of the code, while retaining freedom of programming on the untyped parts, and in turn enables initial untyped prototypes to smoothly evolve into an efficient and robust program.

In [109] we provided a formal account of like types in a core object calculus and evaluated their applicability in the context of a new scripting language, called Thorn, developed at Purdue and IBM.

Contribution 2.
Tool support for semantics: Ott and Lem

Precise definitions of programming languages are needed for two main reasons: to clarify their design, for compiler writers and users, and to reason about programs. Doing this for a full language, however, is a rarely achieved challenge, partly because the mathematical artefacts needed are too large to work on without tool support. In 2006 Peter Sewell and myself designed and implemented the Ott tool for language definitions, compiling them to Coq, HOL, Isabelle/HOL, and LaTeX [92, 91]. Ott has been used successfully in several research projects, like the formalisation of a Java Module System, a provably sound semantics for a large OCaml core, and the work on the GHC and LLVM internal languages. The current version of Ott is about 24000 lines of OCaml; the tool (BSD licence) is available from `http://moscova.inria.fr/~zappa/software/ott`.

Ott excels at writing specifications of programming languages (i.e., inductive relations over syntax), but falls short as a general-purpose specification language. For instance, it is unsuited to define the semantics of concurrent memory accesses of multiprocessor architecture object of Contribution 4. With Scott Owens and Peter Sewell we have recently worked on a novel prototype system, called Lem, specifically designed to support pragmatic engineering of definitions of realistic computer systems. A preliminary release of Lem is described in [71] and available from `http://www.cl.cam.ac.uk/~so294/lem`.

Contribution 3.
End-to-end proofs of concurrent imperative programs

Reasoning about concurrency, if done naively, can greatly complicate program verification tasks. In 2006, Andrew Appel (Princeton U., USA), Aquinas Hobor (Singapore U.), and myself, studied mathematical ways to factor the concurrent part of a programming language from the sequential part; in particular we designed a version of concurrent separation logic adapted to reason about Cminor, a dialect of the C programming language, used by the CompCert certified compiler [57, 60]. This was part of a major research project (`http://www.cs.princeton.edu/~appel/cminor`) to connect machine-verified source programs in sequential and concurrent programming languages to machine-verified optimising compilers. Our results [46] were an initial step toward this ambitious goal, and motivated my investigations on the semantics of multiprocessor machine code, described in Contribution 4.

Contribution 4.
The semantics of multiprocessor machine code

In 2007, with Sewell et al., I started a project that aims at studying the semantics of multiprocessor machine code. Contrarily to what is usually assumed by research on concurrency, memory accesses of real multiprocessors may be reordered in various constrained ways, with

different processors observing the actions of others in inconsistent orders, so one cannot reason about the behaviour of such programs in terms of an intuitive notion of global time. This fundamental problem is exacerbated by the fact that the vendor specifications of processor architectures, e.g. by Intel and AMD (x86), IBM (Power), and ARM, are informal prose documents, with many ambiguities and omissions. We have developed rigorous semantics for multiprocessor programs above the x86 architecture [81, 72, 90], and Power and ARM [9, 80, 79] (although I contributed to the Power and ARM specifications only in an initial phase). Each covers the relaxed memory model, instruction semantics, and instruction decoding, for fragments of the instruction sets, and is mechanised in HOL or Lem. A complete documentation of the project, including papers, reports, and the tools we developed, is available from `http://www.cl.cam.ac.uk/~pes20/weakmemory`.

Contribution 5. *Compilers for Relaxed-Memory Concurrency: testing and validation*

Compilers sometimes generate correct sequential code but break the concurrency memory model of the programming language: these subtle compiler bugs are observable only when the miscompiled functions interact with concurrent contexts, making them particularly hard to detect. With Pankaj Pawan (Master Student from IIT Kanpur, India) and Robin Morisset (Master Student from ENS), I studied how to reduce the hard problem of hunting concurrency compiler bugs in widely used C compilers to differential testing of sequential code and build a tool that puts this strategy to work. In the process we proved correct a theory of sound optimisations in the C11/C++11 memory model, covering most of the optimisations we have observed in real compilers and validating the claim that common compiler optimisations are sound in the C11/C++11 memory model [68]. Our `cmmtest` tool is available from `http://www.di.ens.fr/~zappa/projects/cmmtest`. It identified several mistaken write introductions and other unexpected behaviours in the latest release of the `gcc` compiler.

Previously, with Jaroslav Ševčik, Viktor Vafeiadis, Suresh Jagannathan, and Peter Sewell, I studied the semantic design and verified compilation of a C-like programming language for concurrent shared-memory computation above x86 multiprocessors. The design of such a language is made surprisingly subtle by several factors: the relaxed-memory behaviour of the hardware, the effects of compiler optimisation on concurrent code, the need to support high-performance concurrent algorithms, and the desire for a reasonably simple programming model. In turn, this complexity makes verified compilation both essential and challenging. We defined a concurrent relaxed-memory semantics for ClightTSO, an extension of CompCert's Clight in which the processor memory model is exposed for high-performance code, and, building on CompCert, we implemented and validated with correctness proofs a certifying compiler from ClightTSO to x86. Papers describing our approach have been published in POPL and SAS [84, 104], and in JACM [85] while the development is available from `http://www.cl.cam.ac.uk/~pes20/CompCertTSO`.

Contribution 6. *Formal properties of audit logs*

Many protocols rely on audit trails to allow an impartial judge to verify a posteriori some property of a protocol run. However, in current practice the choice of what data to log is left to the programmer's intuition, and there is no guarantee that it constitutes enough evidence.

Between 2007 and 2010 I pursued this direction of research in collaboration with Cédric Fournet (MSR) and Nataliya Guts (PhD student at the INRIA-MSR joint centre), at the INRIA-MSR joint centre. In a first work [40], we formalised audit logs for a sample optimistic scheme, the value commitment. Then we studied a general scheme to generate audit trails. Given an F# (a dialect of OCaml) program that implements some protocol, we showed that the expected auditable properties can be specified and verified using a type system equipped with refinement types. We implemented our approach using the F7 typechecker, and we tested several protocols, including a full-scale auction-like protocol programmed in ML [44, 43].

Contribution 7. *Theory of higher order processes*

My early work was in concurrency theory, on the behavioural theory of higher-order processes. In 2005 I published two articles that present all my contributions to this field [30, 64], in particular a full-abstraction result for the calculus of Mobile Ambients.

Chapter 1

Shared memory, an elusive abstraction

In which we show that shared memory does not exist, but should.

Problem Most programmers (and most researchers) assume that memory is sequentially consistent: that accesses by multiple threads to a shared memory occur in a global-time linear order. Real multiprocessors, however, use sophisticated techniques to achieve high-performance: store buffers, hierarchies of local cache, speculative execution, etc. These are not observable by single-threaded programs, but in multithreaded programs different threads may see subtly different views of memory. Such machines exhibit *relaxed*, or *weak*, *memory models*. For example, on standard Intel or AMD x86 processors, given two memory locations x and y (initially holding 0), if two processors Proc 0 and Proc 1 respectively write 1 to x and y and then read from y and x, as in the program below, it is possible for both to read 0 *in the same execution*. Is is easy to check that this result cannot arise from any interleaving of the reads and writes of the two processors.

Proc 0	Proc 1
MOV [x]←1 MOV EAX←[y]	MOV [y]←1 MOV EBX←[x]
Allowed Final State: Proc 0:EAX=0 ∧ Proc 1:EBX=0	

Microarchitecturally, one can view this as a visible consequence of *write buffering*: each processor effectively has a FIFO buffer of pending memory writes (to avoid the need to block while a write completes), so the reads from y and x can occur before the writes have propagated from the buffers to main memory.

Such optimisations (of which this is a particularly simple example) destroy the illusion of sequential consistency, making it impossible (at this level of abstraction) to reason in terms of an intuitive notion of global time.

To describe what programmers can rely on, processor vendors document *architectures*. These are loose specifications, claimed to cover a range of past and future actual processors, which should reveal enough for effective programming, but without unduly constraining future processor designs. In practice, however, they are typically informal-prose documents, e.g. the Intel 64 and IA-32 Architectures SDM [5], the AMD64 Architecture Programmer's Manual [4], or the Power ISA specification [6] (SPARC and Itanium have somewhat clearer semi-formal memory models). Informal prose is a poor medium for loose specification of subtle properties, and, as we shall see, such documents are often ambiguous, sometimes

incomplete (too weak to program above), and sometimes unsound (forbidding behaviour that the actual processors allow). Moreover, one cannot test programs above such a vague specification (one can only run programs on particular actual processors), and one cannot use them as criteria for testing processor implementations.

Further, different processor families (Intel 64/IA-32 and AMD64, PowerPC, SPARC, Alpha, Itanium, ARM, etc.) allow very different reorderings, and an algorithm that behaves correctly above one may be incorrect above another.

The Dream For there to be any hope of building reliable multiprocessor software, system programmers need to understand what relaxed-memory behaviour they *can* rely on. In an ideal world each architecture would define a mathematically precise programmer's model, to inform the intuition of systems programmers, to provide a sound foundation for rigorous reasoning about multiprocessor programs, and to give a clear correctness criterion for hardware.

1.1 A Short Tour of Some Real-World Memory Models, circa 2009

We got interested in weak-memory models around 2007 and we started by reading the processor documentation and related resources. Below we review the specifications of some recent, widely used, processors, focussing on the time frame between 2007 and 2009.

Intel 64/IA32 and AMD64 There have been several versions of the Intel and AMD documentation, some differing radically; we contrast them with each other, and with our knowledge of the behaviour of the actual processors.

Pre-IWP (before Aug. 2007) Early revisions of the Intel SDM (e.g. rev-22, Nov. 2006) gave an informal-prose model called 'processor ordering', unsupported by any litmus-test examples. It is hard to give a precise interpretation of this description, as illustrated by the animated discussion between Linux kernel developers on how to correctly implement spinlocks [1], where a simple programming question turns into a microarchitectural debate.

IWP/AMD64-3.14/x86-CC In August 2007, an Intel White Paper (IWP) [3] gave a somewhat more precise model, with 8 informal-prose principles supported by 10 litmus tests. This was incorporated, essentially unchanged, into later revisions of the Intel SDM (including rev.26–28), and AMD gave similar, though not identical, prose and tests [2]. These are essentially causal-consistency models. They allow independent readers to see independent writes (by different processors to different addresses) in different orders, as in the IRIW example below [22]):

Proc 0	Proc 1	Proc 2	Proc 3
MOV [x]←$1	MOV [y]←$1	MOV EAX←[x] MOV EBX←[y]	MOV ECX←[y] MOV EDX←[x]
Allowed Final State: Proc 2:EAX=1 ∧ Proc 2:EBX=0 ∧ Proc 3:ECX=1 ∧ Proc 3:EDX=0			

but require that, in some sense, causality is respected: *"P5. In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility)"*. These were the basis for our x86-CC model [81], for which a key issue was giving a reasonable interpretation to this "causality". Apart from that, the informal specifications were reasonably unambiguous — but they turned out to have two serious flaws.

First, they are arguably rather weak for programmers. In particular, they admit the IRIW behaviour above but, under reasonable assumptions on the strongest x86 memory barrier, MFENCE, adding MFENCEs would not suffice to recover sequential consistency [81, §2.12]. Here the specifications seem to be much looser than the behaviour of implemented processors: to the best of our knowledge, and following some testing, IRIW is not observable in practice. It appears that some JVM implementations depend on this fact, and would not be correct if one assumed only the IWP/AMD64-3.14/x86-CC architecture [34].

Second, more seriously, they are unsound with respect to current processors. The following example, due to Paul Loewenstein [63], shows a behaviour that is observable (e.g. on an Intel Core 2 duo), but that is disallowed by x86-CC, and by any interpretation we can make of IWP and AMD64-3.14.

Proc 0	Proc 1
MOV [x]←1 MOV EAX←[x] MOV EBX←[y]	MOV [y]←2 MOV [x]←2
Allowed Final State: Proc 0:EAX=1 ∧ Proc 0:EBX=0 ∧ [x]=1	

To see why this may be allowed by multiprocessors with FIFO write buffers, suppose that first the Proc 1 write of [y]=2 is buffered, then Proc 0 buffers its write of [x]=1, reads [x]=1 from its own write buffer, and reads [y]=0 from main memory, then Proc 1 buffers its [x]=2 write and flushes its buffered [y]=2 and [x]=2 writes to memory, then finally Proc 0 flushes its [x]=1 write to memory.

Intel SDM rev-29 (Nov. 2008) and rev-30 (Mar. 2009) Subsequent x86 vendor specifications, as revisions 29 and 30 of the Intel SDM (these are essentially identical, and we are told that there will be a future revision of the AMD specification on similar lines) are in a similar informal-prose style to previous versions, again supported by litmus tests, but are significantly different from IWP/AMD64-3.14/x86-CC. First, the IRIW final state above is forbidden [Example 7-7, rev-29], and the previous coherence condition: *"P6. In a multiprocessor system, stores to the same location have a total order"* has been replaced by: *"P9. Any two stores are seen in a consistent order by processors other than those performing the stores"*.

Second, the memory barrier instructions are now included, with *"P11. Reads cannot pass LFENCE and MFENCE instructions"* and *"P12. Writes cannot pass SFENCE and MFENCE instructions"*.

Third, same-processor writes are now explicitly ordered (we regarded this as implicit in the IWP *"P2. Stores are not reordered with other stores"*): *"P10. Writes by a single processor are observed in the same order by all processors"*.

This specification appears to deal with the unsoundness, admitting the behaviour pointed out by Lowenstein, but, unfortunately, it is still problematic. The first issue is, again, how

to interpret "causality" as used in P5. The second issue is one of weakness: the new P9 says nothing about observations of two stores by those two processors themselves (or by one of those processors and one other). This is illustrated by the example below, from [73].

Proc 0	Proc 1
MOV [x]←1 MOV EAX←[x]	MOV [x]←2 MOV EBX←[x]
Forbid: 0:EAX=2 ∧ 1:EBX=1	

This has final states that were not allowed in x86-CC, and we would be surprised if they were allowed by any reasonable implementation (they are not allowed in a pure write-buffer implementation). We have not observed them on actual processors, and programming above a model that permitted them would be problematic. However, rev-29 appears to allow them.

Following this, we proposed an *x86-TSO* model [73, 90], similar to the total store ordering model of SPARCv8 but adapted to handle x86-specific features. It is interesting to summarise the key litmus-test differences:

	IWP/x86-CC	rev-29	x86-TSO	actual processors
IRIW	allowed	forbidden	forbidden	not observed
Lowenstein	forbidden	allowed	allowed	observed
OSS09a	forbidden	allowed	forbidden	not observed

Power and ARM In the most common programming scenario, Power and ARM have weaker models than x86, but are similar to each other. The following discussion is based on the Power ISA Version 2.05 specification [6] (applicable to POWER6 and POWER5 processors) and the ARM Architecture Reference Manual [13] (applicable to ARMv7 processors). A key concept in these informal-prose architecture specifications is that of accesses being "performed" (Power) or "observed" (ARM) with respect to processors. "Performed" is defined as follows [6, p.408]:

> A load or instruction fetch by a processor or mechanism (P1) is performed with respect to any processor or mechanism (P2) when the value to be returned by the load or instruction fetch can no longer be changed by a store by P2. A store by P1 is performed with respect to P2 when a load by P2 from the location accessed by the store will return the value stored (or a value stored subsequently).

This is used in the informal semantics of barriers (`sync`, `lwsync`, `eieio`, `DMB`, `DSB`), and of dependencies, e.g. [6, Book II,§1.7.1,p.413]:

> If a *Load* instruction depends on the value returned by a preceding *Load* instruction, the corresponding storage accesses are performed in program order with respect to any processor or mechanism.

Such a definition of "performed" does not lend itself to a direct formalisation. First, it implicitly refers to a notion of global time. That can be easily solved, as we are only

concerned with whether one access is performed before or after another. Second, more seriously, it is subjunctive: the first clause refers to a hypothetical store by P2, and the second to a hypothetical load by P2. A memory model should define whether a particular execution is allowed, and it would be awkward in the extreme to define this in terms of executions modified by adding such hypothetical accesses.

Several initially-plausible interpretations turn out to be too weak or unsound. One could adopt view orders, per-processor orders capturing when events become visible to each processor in its local view of time, and consider an access to be "performed" from the point where it appears in the view order. Intuitively, the visible events are those that influence the processor's behaviour, and we defined a preliminary such model [9] in which we do not include other processors' loads in view orders. Doing this naively gives too weak a semantics for barriers, Alternatively, one can include other processor's read events. Again, doing so naively would be wrong: the Load/Load dependency text above suggests that the program order of two loads performed on the same processor is preserved even if they are not interleaved with a barrier instruction, but we have observed non-sequentially consistent behaviour in a variant of the IRIW example above if there is an address dependency between each pair of loads, but not if there is a barrier between each pair of loads. Adir et al. [7] give a more complex model including some foreign loads in view orders, for an earlier PowerPC specification. However, none of these models accounts completely for the Power 2.05 barriers, capturing the rather subtle differences between `sync` and its lighter alternative `lwsync`, and identifying exactly what barriers are required to regain sequential consistency.

If it is this hard to give a consistent interpretation to the architecture documentation, one has to wonder whether correct low-level code could be written based on it, without additional knowledge of the processor implementations.

1.2 Towards Rigorous Memory Models

What, then, is the way forward? Existing real-world memory models cannot be completely trusted, and, although there exists an extensive literature on relaxed memory models, most of it does not address real processor semantics, or is not based on rigorous mathematical models. We argue that a specification for a multiprocessor or programming-language memory model should satisfy several stringent criteria.

First, it should be *precise*. It should state unambiguously what is and is not permitted. This must be with mathematical precision, not in informal prose — experience shows that the latter, even when written with great care, is prone to ambiguity and omission. The mathematical definitions should be normative parts of the architecture or language standard. Ideally, these mathematical definitions should be expressed and mechanised in a proof assistant, supporting mechanical type-checking and independently machine-checked proofs of metatheoretic results.

Second, it should be *testable*. Given a program, it should be possible to compute (a) whether some particular candidate execution is admitted by the memory model, and (b) the set of all such admissible executions (up to a certain length). Ideally, the algorithms for these should be derived automatically from the statement of the model, or from some provably-equivalent alternative characterisation thereof, to reduce the possibility of translation error.

Third, it should be *accurate with respect to implementations* (of processors or compilers). Given the above, this can be tested empirically, as it is easy to run programs on real hardware or a real compiler-and-hardware combination. The model should allow all the behaviours observed in practice. This testing, of the behaviour of the real implementations against the published rigorous specification, should be part of the normal development process of the processors or compilers. In principle, accuracy can also be established by proof, showing that the semantics is an accurate abstraction of a microarchitecture or compiler. That would be highly desirable, but very challenging: for modern multiprocessors and compilers, such a model would be very large, and typically also proprietary.

Fourth, it should be *loose enough for future implementations*: the range of permitted behaviour should be wide enough to allow reasonable developments in implementations. However, this point should not be over-emphasised at the expense of the others, as seems to have often happened in the past.

Fifth, it should be *strong enough for programmers*. A well-specified model should constrain the behaviours enough that reasonable programs can be shown (by informal reasoning, proof, or exhaustive symbolic emulation) to have their intended behaviour, without relying on any other knowledge of the implementations. A different and complementary approach is to formally prove metatheoretic results about the model, including data-race-freedom properties; these proofs are subtle and should be mechanised.

Sixth, it should be *integrated with the semantics of the rest of the system* (describing the behaviour of the processor instructions or of the phrases of the programming language). Memory models are typically presented in isolation, and this makes it all too easy to gloss over important details.

Lastly, it should be *accessible*, to concurrent programmers, hardware architects, language designers and implementors, and builders of verification tools, as the interface between these four groups. For that it should be expressed in straightforward logic, not some exotic specialised formalism, and should be extensively annotated so that it can be read by non-mathematicians. Having a precise mathematical specification will make it easier to write self-consistent tutorial documents. For processors, where possible, it seems desirable to have both an operational (or abstract machine) model and a provably equivalent axiomatic model; the former are often more comprehensible and the latter more useful for some metatheory, and an equivalence proof may detect errors and inconsistencies. However, operational models should not involve more microarchitectural detail than is necessary: it should be clearly understood that these are specifications of the programmer-visible behaviour, not descriptions of any actual microarchitecture.

1.3 State of the Art and the Way Forward

Since 2007 we have developed semantics for multiprocessor programs above the x86 architecture [81, 72, 90], and Power and ARM [9, 80, 79] (although I contributed to the Power and ARM specifications only in an initial phase). Each covers the relaxed memory model, instruction semantics, and instruction decoding for fragments of the instruction sets, and is mechanised in HOL (and more recently in the LEM language, discussed in Chapter 3).

Focussing on x86, at the time of writing there is consensus that the memory model for user-mode code is correctly described by our x86-TSO model. This model is *precise*, it gives equivalent operational and axiomatic definitions expressed in mathematical language, as a transition system the former and as constraints on relations the latter. It is *testable* as computing all the executions of a program is a state exploration problem; in particular the axiomatic model has been implemented in our `memevents` tool (today superseded by the `herd` tool of Maranget and Alglave) which computes the set of all valid executions of simple test programs. *Accuracy with respect to implementations* has been extensively tested by comparing the output of our `litmus` tool, that stress-tests a processor against a given litmus test, and `memevents`. It is hard to assess if it is *loose enough for future implementations* (at least it seems compatible with the vendor intentions) but it is definitely *strong enough for programmers*: several subtle concurrent algorithms have been implemented, and even proved correct, on a TSO memory and a wide range of tools (from data-race detectors to compilers) can reason about TSO executions. It is easy to build web-tools to animate possible transitions and make the model *accessible*; we have been told that engineers at IBM were suddenly interested in the ARM/Power model once we made the `ppcmem` tool available on the web (`http://www.cl.cam.ac.uk/~pes20/ppcmem/` — it should be pointed out that the web application is mostly generated from (and kept in synch with) the Lem specification via the `js_of_ocaml` OCaml to JavaScript compiler).

However much remains to be done. Our models deal with *coherent write-back memory*, which is used by most user and OS code, but assume no exceptions, no interrupts, no misaligned or mixed-size accesses, no 'non-temporal' operations, no device memory, no self-modifying code, and no page-table changes. These features should be rigorously modeled, and this is a non-trivial extension of our work. For instance, in our x86-TSO model the LFENCE and SFENCE memory barriers are no-ops; while in some cases not covered by our formalisation they likely are not. Also, our models cover only a limited subset of the instruction semantics. Several projects ([60, 69, 53] to cite just a few) formalise subsets of the x86 instruction set large enough for verification of some realistic code, but it is a shame that we do not yet dispose of a comprehensive formalisation of the x86 ISA; even worse, none integrates with the processors' relaxed-memory concurrency semantics, virtual memory or inter-processor interrupts. Additionally, an ISA model should be executable and should be designed to explore and simulate possible whole-system behaviours, as a typical application would be to use them as fast oracles to quickly check whether some behaviour is permitted or not. Preliminary experiments toward building "daemonic emulators", weighted to randomly expose allowed but rare relaxed behaviours, suggest that these can be effective and lightweight tools to debug concurrent code.

Chapter 2

Thread-wise reasoning on shared memory concurrent systems

In which we show how some problems can be made tractable by performing thread-wise reasoning on shared memory concurrent systems, but conclude that the general case of compositional reasoning is still open despite 25 years of research in concurrency theory.

2.1 Hunting Concurrency Compiler Bugs

Problem Consider the C program in Figure 2.1. Can we guess its output? This program spawns two threads executing the functions `th_1` and `th_2` and waits for them to terminate. The two threads share two global memory locations, `g1` and `g2`, but a careful reading of the code reveals that the inner loop of `th_1` is never executed and `g2` is only accessed by the second thread, while `g1` is only accessed by the first thread. According to the C11/C++11 standards [18] this program should always print `42` on the standard output: the two threads do not access the same variable in conflicting ways, the program is data-race free, and its semantics is defined as the interleavings of the actions of the two threads. However if we compile the above code with the version 4.7.0 of `gcc` on an x86_64 machine running Linux, and we enable some optimisations with the `-O2` flag, as in

```
$  g++ -std=c++11 -lpthread -O2 -S foo.c
```

then, sometimes, the compiled code prints `0` to the standard output. This unexpected outcome is caused by a subtle compiler bug. If we inspect the generated assembly we discover that `gcc` saves and restores the content of `g2`, causing `g2` to be overwritten with `0`:

```
th_1:
     movl  g1(%rip), %edx    # load g1 (1) into edx
     movl  g2(%rip), %eax    # load g2 (0) into eax
     testl %edx, %edx        # if g1 != 0
     jne   .L2               #   jump to .L2
     movl  $0, g2(%rip)
     ret
.L2: movl  %eax, g2(%rip)    # store eax (0) into g2
     xorl  %eax, %eax        # return 0 (NULL)
     ret
```

```
#include <stdio.h>
#include <pthread.h>

int g1 = 1; int g2 = 0;

void *th_1(void *p) {
  int l;
  for (l = 0; l != 4; l++) {
    if (g1 != 0) return NULL;
    for (g2 = 0; g2 >= 26; ++g2)
      ;
  }
}

void *th_2(void *p) {
  g2 = 42;
  printf("%d\n",g2);
}

int main() {
  pthread_t th1, th2;
  pthread_create(&th1, NULL, th_1, NULL);
  pthread_create(&th2, NULL, th_2, NULL);
  (void)pthread_join (th1, NULL);
  (void)pthread_join (th2, NULL);
}
```

Figure 2.1: `foo.c`, a concurrent program miscompiled by `gcc` 4.7.0.

This optimisation is sound in a sequential world because the extra store always rewrites the initial value of `g2` and the final state is unchanged. However, as we have seen, this optimisation is unsound in a concurrent context as that provided by `th_2` and the C11/C++11 standards forbid it.

How can we build assurance in widely used implementations of C and C++ such as `gcc` and `clang`?

The Dream In an ideal world mainstream compilers would be bug-free. In practice differential random testing proved successful at hunting compiler bugs. The idea is simple: a test harness generates random, *well-defined*, source programs, compiles them using several compilers, runs the executables, and compares the outputs. The state of the art is represented by the Csmith tool by Yang, Chen, Eide and Regehr [110], which over the last four years has discovered hundreds of bugs in widely used compilers as `gcc` and `clang`. However this work cannot find concurrency compiler bugs like the one we described above: despite

being miscompiled, the code of `th_1` still has correct behaviour in a sequential setting. A step toward fulfilling the dream is to have a tool analogous to Csmith that via differential random testing of C and C++ compilers detects and reports concurrency compiler bugs.

Random testing for concurrency compiler bugs A naive approach to extend differential random testing to concurrency bugs would be to generate *concurrent* random programs, compile them with different compilers, record *all* the possible outcomes of each program, and compare these sets. This works well in some settings, such as the generation and comparison of litmus tests to test hardware memory models; see for instance the work by Alglave et al. [10]. However this approach is unlikely to scale to the complexity of hunting C11/C++11 compiler bugs. Concurrent programs are inherently *non-deterministic* and optimisers can compile away non-determism. In an extreme case, two executables might have disjoint sets of observable behaviours, and yet both be correct with respect to a source C11 or C++11 program. The correctness check cannot just compare the final checksum of the different binaries but must check that all the behaviours of a compiled executable are allowed by the semantics of the source program. The Csmith experience suggests that surprisingly large program sizes (∼80KB) are needed to maximise the chance of hitting corner cases of the optimisers; at the same time they must exhibit subtle interaction patterns (often unexpected, as in the example above) while being well-defined (in particular data-race free). Capturing the set of all the behaviours of such large scale concurrent programs is tricky as it can depend on rare interactions in the scheduling of the threads, and computing all the behaviours allowed by the C11/C++11 semantics is even harder.

Despite this, we show that differential random testing can be used successfully for hunting concurrency compiler bugs. First, C and C++ compilers must support separate compilation and the concurrency model allows any function to be spawned as an independent thread. As a consequence compilers must always assume that the *sequential* code they are optimising can be run in an *arbitrary concurrent context*, subject only to the constraint that the whole program is well-defined (race-free on non-atomic accesses, etc.), and can only apply optimisations which are sound with respect to the concurrency model. Second, it is possible to characterise which optimisations are correct in a concurrent setting by observing how they eliminate, reorder, or introduce, memory accesses in the traces of the sequential code with respect to a reference trace. Combined, these two remarks imply that testing the correctness of compilation of concurrent code can be reduced to validating the traces generated by running optimised sequential code against a reference (unoptimised) trace for the same code.

We illustrate this idea with program `foo.c` from Figure 2.1. Traces only report accesses to global (potentially shared) memory locations because optimisations affecting only the thread-local state cannot induce concurrency compiler bugs. The reference trace on the left for `th_1` initialises `g1` and `g2` and loads the value `1` from `g1`:

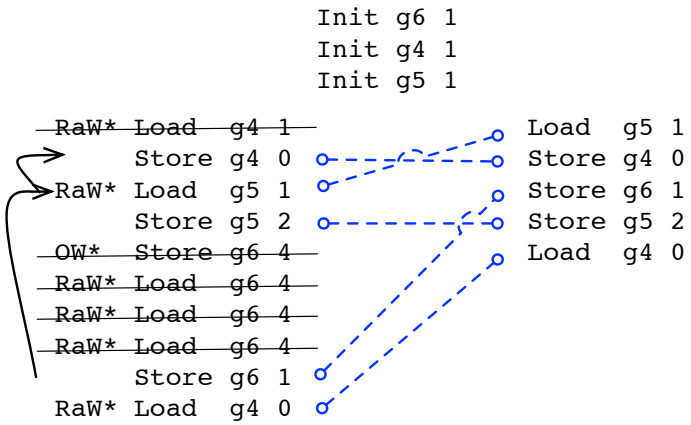
```
Init  g1 1
Init  g2 0
Load  g1 1
```

```
Init  g1 1
Init  g2 0
Load  g1 1
Load  g2 0
Store g2 0
```

```
const unsigned int g3 = 0UL;
long long g4 = 0x1;
int g6 = 6L;
unsigned int g5 = 1UL;

void f(){
  int *l8 = &g6;
  int l36 = 0x5E9D070FL;
  unsigned int l107 = 0xAA37C3ACL;
  g4 &= g3;
  g5++;
  int *l102 = &l36;
  for (g6 = 4; g6 < (-3); g6 += 1);
  l102 = &g6;
  *l102 = ((*l8) && (l107 << 7)*(*l102));
}
```

This randomly generated function generates the following traces if compiled with `gcc -O0` and `gcc -O2`.



All the events in the optimised trace can be matched with events in the reference trace by performing valid eliminations and reorderings of events. Eliminated events are ~~struck-off~~ while the reorderings are represented by the arrows.

Figure 2.2: successful matching of reference and optimised traces

while the trace of the `gcc -O2` generated code on the right instead performs an extra load and store to `g2`. Since arbitrary store introduction is provably incorrect in the C11/C++11 concurrency model we can detect that a miscompilation happened. Figure 2.2 shows another example, of a randomly generated C function together with its reference trace and an optimised trace. In this case it is possible to match the reference trace (on the left) against the optimised trace (on the right) by a series of sound eliminations and reordering of actions.

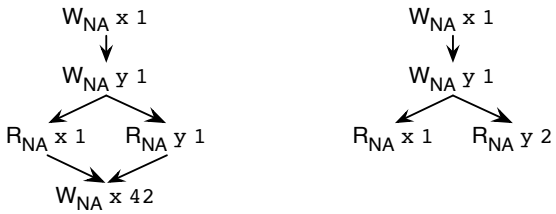
A theory of sound optimisations in the C11/C++11 memory model This approach to compiler testing crucially relies on a compositional theory of sound optimisations over executions of C11/C++11 programs: we must characterise which thread-local optimisations are sound in an arbitrary non-racy concurrent context.

Compiler optimisations are usually described as program transformations over an intermediate representation of the source code; a typical compiler performs literally hundreds of optimisation passes. Although it is possible to prove the correctness of individual transformations, this presentation does not lend itself to a thorough characterisation of what program transformations are valid.

In a source program each thread consists of a sequence of *instructions*. During the execution of a program, any given static instruction may be iterated multiple times (for example due to looping) and display many behaviours. For example reads may read from different writes and conditional branches may be taken or not. We refer to each such instance of the dynamic execution of an instruction as an *instruction instance*. More precisely, each instruction instance performs zero, one, or more shared memory accesses, which we call *actions*. We account for all the differing ways a given program can execute by identifying a source program with the set of sets of all the actions (annotated with additional information as described below) it can perform when it executes in an arbitrary context. We call the set of the actions of a particular execution an *opsem* and the set of all the opsems of a program an *opsemset*. For instance, the snippet of code below:

```
x = 1; y = 1; if (x == y){x = 42;}
```

has, among others, the two opsems below:



The opsem on the left corresponds to an execution where the reads read the last values written by the thread itself; the opsem on the right accounts for an arbitrary context that concurrently updated the value of `y` to 2. Nodes represent actions and black arrows show the *sequenced-before relation*, which orders actions (by the same thread) according to their program order. The sequenced-before relation is not total because the order of evaluation of the arguments of functions, or of the operands of most operators, is underspecified in C and C++. The memory accesses are non-atomic, as denoted by the NA label.

We can then characterise the effect of arbitrary optimisations of source code directly on opsemsets. On a given opsem, the effect of any transformation of the source code is to eliminate, reorder, or introduce actions. If the optimiser performs constant propagation, the previous code is rewritten as:

```
x = 1; y = 1; if (1 == 1){x = 42;}
```

and its unique opsem is depicted here on the right. This opsem can be obtained

```
for (i=0; i<2; i++) {
  z = z + y + i;
  x = y;
}
```

⇒

```
t = y; x = t;
for (i=0; i<2; i++) {
  z = z + t + i;
}
```

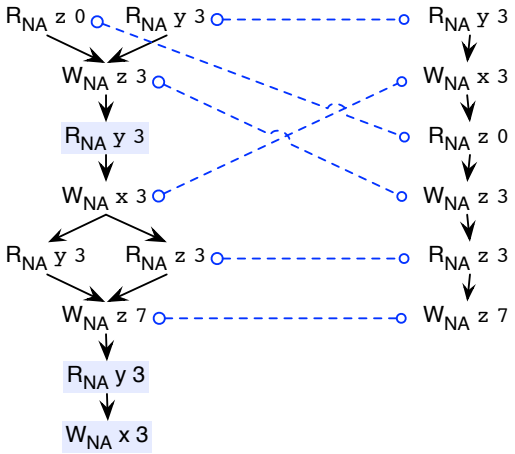
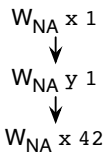


Figure 2.3: effect of loop invariant code motion (LIM) on an opsem

from the one above on the left by eliminating the two read actions. A more complex example is shown in Figure 2.3, where the loop on the left is optimised by LIM. The figure shows opsems for the initial state `z=0`, `y=3` assuming that the code is not run in parallel with an interfering context. Here the effect of the LIM optimisation is not only to remove some actions (in blue) but also to reorder the write to `x`.



An opsem captures a possible execution of the program, so by applying a transformation to an opsem we are actually optimising one particular execution. Lifting pointwise this definition of *semantic transformations* to opsemsets enables optimising all the execution paths of a program, one at a time, thus abstracting from actual source program transformation.

Overview of the C11/C++11 memory model To understand the theory of sound optimisations we give a high-level overview of the C11/C++11 memory model as formalised by Batty et al. [17], defining opsems, opsemsets and executions.

Let l range over locations (which are partitioned into non-atomic and atomic), v over values and $tid \in \{1..n\}$ over thread identifiers. We consider the following actions:

$$\begin{aligned}
&\mathsf{mem_ord}, \mu \quad ::= \quad \mathsf{NA} \mid \mathsf{SC} \mid \mathsf{ACQ} \mid \mathsf{REL} \mid \mathsf{R/A} \mid \mathsf{RLX} \\
&\phi \quad ::= \quad \mathsf{R}_\mu\, l\, v \mid \mathsf{W}_\mu\, l\, v \mid \mathsf{Lock}\, l \mid \mathsf{Unlock}\, l \mid \mathsf{Fence}_\mu \mid \mathsf{RMW}_\mu\, l\, v_1\, v_2 \\
&\mathsf{actions} \quad ::= \quad aid, tid{:}\phi
\end{aligned}$$

The possible actions are loads from and stores to memory, lock and unlock of a mutex, fences, and read-modify-writes of memory locations. Each action is identified by an action identifier aid (ranged over by $r, w, \ldots$) and specifies its thread identifier tid, the location l it affects,

the value read or written v (when applicable), and the memory-order μ (when applicable).[1] In the drawings we omit the action and thread identifiers.

The thread-local semantics identifies a program with a set of *opsems* (ranged over by O): triples $(A, \mathsf{sb}, \mathsf{asw})$ where $A \in P(\mathsf{actions})$ and $\mathsf{sb}, \mathsf{asw} \subseteq A \times A$ are the *sequenced-before* and *additional-synchronised-with* relations. Sequenced-before (denoted sb) was introduced above; it is transitive and irreflexive and only relates actions by the same thread; *additional-synchronised-with* (denoted asw) contains additional edges from thread creation and thread join, and in particular orders initial writes to memory locations before all other actions in the execution.

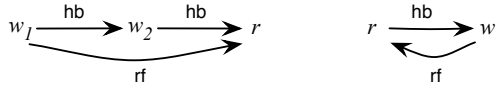
The thread-local semantics assumes that all threads run in an arbitrary concurrent context which can update the shared memory at will. This is modelled by reads taking unconstrained values. We say that a set of opsems S is *receptive* if, for every opsem O, for every read action $r, t{:}\mathsf{R}_\mu\, l\, v$ in the opsem O, for all values v' there is an opsem O' in S which only differs from O because the read r returns v' rather than v, and for the actions that are sequenced-after r. Intuitively a set of opsems is receptive if it defines a behaviour for each possible value returned by each read.

We call the set of all the opsems of a program an *opsemset*, ranged over by P. The thread local semantics ensures that opsemsets are receptive. Opsems and opsemsets are subject to several well-formedness conditions, e.g. atomic accesses must access only atomic locations, which we omit here and can be found in [17]. We additionally require opsemsets to be sb-prefix closed, assuming that a program can halt at any time. Formally, we say that an opsem O' is an sb-prefix of an opsem O if there is an injection of the actions of O' into the actions of O that behaves as the identity on actions, preserves sb and asw, and, for each action $x \in O'$, whenever $x \in O$ and $y <_{\mathsf{sb}} x$, it holds that $y \in O'$.

Executions The denotation of each thread in an opsem is agnostic to the behaviour of the other threads of the program: the thread-local semantics takes into account only the structure of every thread's statements, not the semantics of memory operations. In particular, the values of reads are chosen arbitrarily, without regard for writes that have taken place. The memory model filters inconsistent opsems by constructing additional relations and checking the resulting candidate executions against the axioms of the model. For this an *execution witness* (denoted by W) for an opsem specifies an interrelationship between memory actions of different threads via three relations: *reads-from* (rf) relates a write to all the reads that read from it; the *sequential consistent order* (sc) is a total order over all SC actions; and *modification order* (mo) – or *coherence* – is the union of a per-location total order over writes to each atomic location. From these, the model infers the relations *synchronises-with* (denoted sw), which defines synchronisation and is described in detail below, and *happens-before* (denoted hb), showing the precedence of actions in the execution. Key constraints on executions depend on the happens-before relation, in particular a *non-atomic read* must not read any write related to it in hb other than its immediate predecessor. This property

[1] we omit *consume* atomics: their semantics is intricate (e.g. happens-before is not transitive in the full model) and at the time of writing no major compiler profits from the their weaker semantics, treating consume as acquire. By general theorems [16], our results remain valid in the full model.

is called *consistent non-atomic read values*, and for writes w_1 and w_2 and a read r accessing the same non-atomic location, the following shapes are forbidden:



For atomic accesses the situation is more complex, and atomic reads can read from hb-unrelated writes.

Happens-before is a partial relation defined as the transitive closure of sb, asw and sw: $\mathsf{hb} = (\mathsf{sb} \cup \mathsf{asw} \cup \mathsf{sw})^+$.

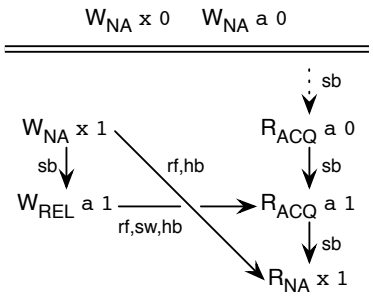
We refer to a pair of an opsem and witness (O, W) as a *candidate execution*. A pair (O, W) that satisfies a list of *consistency predicates* on these relations (including consistent non-atomic read values) is called a *pre-execution*. The model finally checks if none of the pre-executions contain an *undefined behaviour*. Undefined behaviours arise from *unsequenced races* (two conflicting accesses performed by the same thread not related by sb), *indeterminate reads* (an access that does not read a written value), or *data races* (two conflicting accesses not related by hb), where two accesses are *conflicting* if they are to the same address and at least one is a non-atomic write. Programs that exhibit an undefined behaviour (e.g. a data-race) in one of their pre-executions are undefined; programs that do not exhibit any undefined behaviour are called *well-defined*, and their semantics is given by the set of their pre-executions.

Synchronisation Synchronisation between threads is captured by the sw relation. The language provides two mechanisms for establishing synchronisation between threads and enabling race-free concurrent programming: *mutex locks* and *low-level atomics*. The semantics of mutexes is intuitive: the sc relation, part of the witness, imposes a total order over all lock and unlock accesses, and a synchronised-with (sw) edge is added between every unlock action, and every lock of the same mutex that follows it in sc-order. Low-level atomics are specific to C/C++ and designed as an escape hatch to implement high-performance racy algorithms. Atomic operations do not race with each other, by definition, and their semantics is specified by a memory-order attribute. *Sequentially consistent* atomics have the strongest semantics: all SC reads and writes are part of the total order sc (acyclic with hb). An SC read can only read from the closest sc-preceding write to the same location. Although sc and hb must be compatible, sc is not included in hb. Sequentially consistent atomics, as well as release-acquire atomics, generate synchronisation edges, which are included in hb. This is best explained for a classic message-passing idiom. Imagine that one thread writes some (perhaps multi-word) data x and then an atomic flag a, while another waits to see that flag write before reading the data:

```
                          x=0; atomic a=0
x = 1;                      || while (0 == a.load(acq)) {};
a.store(1, rel);            || int r = x;
```

The synchronisation between the release and acquire ensures that the sender's write of x will be seen by the receiver. Below we depict a typical opsem for this program; we represent the asw relation with the double horizontal line: the init actions are asw-before

all other events. The witness has an rf arrow between the write-release and read-acquire on a to justify that the read returns 1. A read between a write-release and a read-acquire generates an sw edge. Since hb includes sb and sw, the write of 1 to x is the last write in the hb order before the read of the second thread, which is then forced to return 1. *Relaxed atomics* instead do not generate synchronisation edges sw; they are only forbidden to read from the future, i.e. from writes later in hb or mo.



Observable behaviour The C11/C++11 memory model does not explicitly define the *observable behaviour* of an execution. We extend the model with a special atomic location called `world`, and model the observable side-effects of the program (e.g., writes on `stdout`) by relaxed writes to that location. The relaxed attribute guarantees that these accesses are totally ordered with each other, as captured by the mo relation. As a result, the *observable behaviour* of a pre-execution is the restriction of the mo relation to the distinguished `world` location. If none of its pre-executions exhibit an undefined behaviour, then the observable behaviour of a program is the set of all observable behaviours of its executions.

Sound Optimisations in the C Memory Model C and C++ are shared-memory-concurrency languages with explicit thread creation and implicit sharing: any location might be read or written by any thread for which it is reachable from variables in scope. It is the programmer's responsibility to ensure that such accesses are race-free. This implies that compilers can perform optimisations that are not sound for racy programs and common thread-local optimisations can still be done without the compiler needing to determine which accesses might be shared.

Sevcik showed that a large class of elimination and reordering transformations are correct (that is, do not introduce any new behaviour when the optimised code is put in an arbitrary data-race free context) in an idealised DRF model [82, 83]. We adapt and extend his results to optimise non-atomic accesses in the C11/C++11 memory model. As we have discussed, we classify program transformations as *eliminations*, *reorderings*, and *introductions* over opsemsets.

In this document we focus on eliminations, criteria for reorderings and introductions can be found in [68]. The semantic elimination transformation is general enough to cover optimisations that eliminate memory accesses based on data-flow analysis, such as common subexpression elimination, induction variable elimination, and global value numbering, including the cases when these are combined with loop unrolling.

Definition 2.1.1 *An action is a* release *if it is an unlock action, an atomic write with memory-order REL or SC, a fence or read-modify-write with memory-order REL, R/A or SC.*

Semantically, release actions can be seen as potential sources of sw edges. The intuition is that they "release" permissions to access shared memory to other threads. Symmetrically, acquire actions can be seen as potential targets of sw edges; the intuition is that they "acquire" permissions to access shared memory from other threads.

Definition 2.1.2 *An action is an* acquire *if it is a lock action, or an atomic read with memory-order ACQ or SC, or a fence or read-modify-write with memory order ACQ, R/A or SC.*

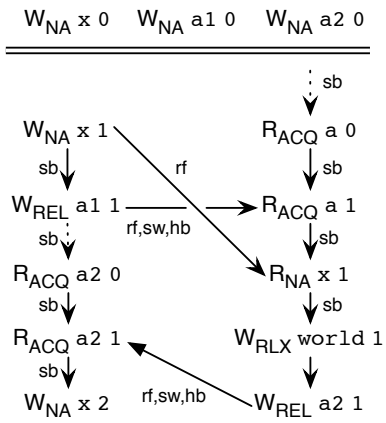
A key concept is that of a *same-thread release-acquire pair*:

Definition 2.1.3 *A* same-thread release-acquire pair *(shortened* st-release-acquire pair*) is a pair of actions (r, a) such that r is a release, a is an acquire, and $r <_{\mathsf{sb}} a$.*

Note that these may be to different locations and never synchronise together. To understand the role they play in optimisation soundness, consider the code on the left, running in the concurrent context on the right:

```
                                    x=0; atomic a1,a2=0
x = 1;                          ||  while(0==a1.load(acq)) {};
a1.store(1,rel);                ||  printf("%i",x);
while(0==a2.load(acq)) {};      ||  a2.store(1,rel);
x = 2;                          ||
```

All executions have similar opsems and witnesses, depicted below (we omitted rf arrows from initialisation writes). No consistent execution has a race and the only observable behaviour is printing `1`. Eliminating the first store to `x` (which might appear redundant as `x` is later overwritten by the same thread) would preserve DRF but would introduce a new behaviour where `0` is printed. However, if either the release or the acquire were not in between the two stores, then this context would be racy (respectively between the load performed by the print and the first store, or between the load and the second store) and it would be correct to optimise away the first write. More generally, the proof of the Theorem 2.1.7 below clearly shows that the presence of an intervening same-thread release-acquire is a necessary condition to allow a discriminating context to interact with a thread without introducing data races.



Definition 2.1.4 *A read action $a, t{:}R_{NA}\, l\, v$ is* eliminable *in an opsem O of the opsemset P if one of the following applies:*

Read after Read (RaR): there exists another action $r, t{:}R_{NA}\, l\, v$ such that $r <_{\mathsf{sb}} a$, and there does not exist a memory access to location l or a st-release-acquire pair sb*-between r and a;*

Read after Write (RaW): there exists an action $w, t{:}W_{NA}\, l\, v$ such that $w <_{\mathsf{sb}} a$, and there does not exist a memory access to location l or a st-release-acquire pair sb*-between w and a;*

Irrelevant Read (IR): for all values v' there exists an opsem $O' \in P$ and a bijection f between actions in O and actions in O', such that $f(a) = a', t{:}R_{NA}\, l\, v'$, for all actions $u \in O$ different from a, $f(u) = u$, and f preserves sb *and* asw.

A write action $a, t{:}W_{NA}\, l\, v$ is eliminable *in an opsem O of the opsemset P if one of the following applies:*

Write after Read (WaR): there exists an action $r, t{:}R_l\, v$ such that $r <_{\mathsf{sb}} a$, and there does not exist a memory access to location l or a st-release-acquire pair sb*-between r and a;*

Overwritten Write (OW): there exists another action $w, t{:}W_{NA}\, l\, v'$ such that $a <_{\mathsf{sb}} w$, and there does not exist a memory access to location l or a st-release-acquire pair sb*-between a and w;*

Write after Write (WaW): there exists another action $w, t{:}W_{NA}\, l\, v$ such that $w <_{\mathsf{sb}} a$, and there does not exist a memory access to location l or a st-release-acquire pair sb*-between w and a.*

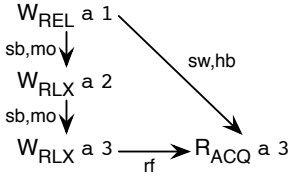
Note that the OW rule is the only rule where the value can differ between the action eliminated and the action that justifies the elimination. The IR rule can be rephrased as "a read in an execution is irrelevant if the program admits other executions (one for each value) that only differ for the value returned by the irrelevant read". We have observed all these cases being performed by the `gcc` and `clang` compilers.

Definition 2.1.5 *An opsem O' is an* elimination *of an opsem O if there exists a injection $f : O' \to O$ that preserves actions,* sb*, and* asw*, and such that the set $O \backslash f(O')$ contains exactly one eliminable action. The function f is called an* unelimination.

To simplify the proof of Theorem 2.1.7 the definition above allows only one elimination at a time (this avoids a critical pair between the rules OW and WaW whenever we have two writes of the same value to the same location), but, as the theorem shows, this definition can be iterated to eliminate several actions from one opsem while retaining soundness. The definition of eliminations lifts pointwise to opsemsets:

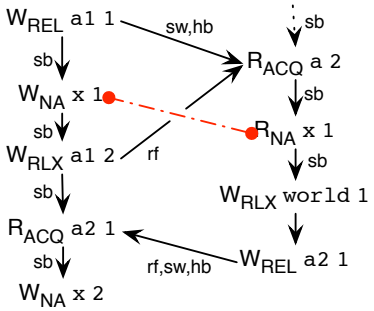
Definition 2.1.6 *An opsemset P' is an* elimination *of an opsemset P if for all opsem $O' \in P'$ there exists an opsem $O \in P$ such that O' is an elimination of O.*

In the previous section we did not describe all the intricacies of the C11/C++11 model but our theory takes all of them into account. For example, a release fence followed by an atomic write behaves as if the write had the REL attribute, except that the sw edge starts from the fence action. Another example is given by *release sequences*: if an atomic write with attribute `release` is followed immediately in mo-order by one or more `relaxed` writes in the same thread (to the same location), and an atomic load with attribute `acquire` reads-from one of these relaxed stores, an sw edge is created between the write release and the load acquire, analogously to the case where the acquire reads directly from the first write. These subtleties must



be taken into account in the elimination correctness proof but do
not invalidate the intervening same-thread release-acquire pair criterion. This follows from a property of the C11/C++11 design that makes every sw edge relate a release action to an acquire action. For instance, in the program below it is safe to remove the first write to `x` as OW, because all discriminating contexts will be necessarily racy:

```
a1.store(1,rel);
x = 1;
a1.store(2,rlx);
while(0==a2.load(acq)) {};
x = 2;
```



We establish that our semantic transformations have the following properties: any execution of the transformed opsemset has the same observable behaviour of some execution of the original opsemset, and the transformation preserves data-race freedom. As C11 and C++11 do not provide any guarantee for racy programs we cannot prove any result about out-of-thin-air value introduction.

Theorem 2.1.7 *Let the opsemset P' be an elimination of the opsemset P. If P is well-defined, then so is P', and any execution of P' has the same observable behaviour of some execution of P.*

We sketch the structure of the proof; details can be found online [32]. Let the opsemset P' be an elimination of the opsemset P, and let (O', W') be an execution of P' (that is, a pair of an opsem and a witness). Since P' is an elimination of P, there is at least one opsem $O \in P$ such that O' is an elimination of O, and an unelimination function f that injects the events of the optimised opsem into the events of the unoptimised opsem. We build the mo and sc relations of the witness of O by lifting the mo$'$ and sc$'$ relations of the witness W': this is always possible because our program transformations do not alter any atomic access. Analogously it is possible to build the rf relation on atomic accesses by lifting the rf$'$ one. To complete the construction of the witness we lift the rf$'$ relation on non-atomic events as well, and complete the rf relation following the case analysis below on the eliminated events in O:

- RaR: if i is a read action eliminated with the RaR rule because of a preceding read action r, and $w <_{\mathsf{rf}} r$, then add $w <_{\mathsf{rf}} i$;
- RaW: if i is a read action eliminated with the RaW rule because of a preceding write action w, then add $w <_{\mathsf{rf}} i$;
- IR: if i is an irrelevant read, and there is a write event to the same location that happens before it, then let w be a write event to the same location maximal with respect to hb and add $w <_{\mathsf{rf}} i$;

- OW: rf is unchanged;
- WaW: if i is a write event eliminated by the WaW rule because of the preceding write event w, then for all actions r such that $w <_{\mathsf{rf}'} r$ and $i <_{\mathsf{hb}} r$, replace $w <_{\mathsf{rf}} r$ by $i <_{\mathsf{rf}} r$;
- WaR: if i is a read event eliminated by the WaR rule, then every read of the same value at the same location, that happens-after i and that either read from a write $w <_{\mathsf{hb}} i$ or does not read from any write, now reads from i.

This completes the construction of the witness W and in turn of the candidate execution (O, W) of P. We must now prove that (O, W) is consistent, in particular that it satisfies *consistent non-atomic read values*, for which the construction has been tailored. This proceeds by a long case disjunction that relies on the following constructions:

- the absence of a release-acquire pair between two accesses a and b in the same thread guarantees the absence of an access c in another thread with $a <_{\mathsf{hb}} c <_{\mathsf{hb}} b$.
- in some cases the candidate execution (O, W) turns out to have conflicting accesses a and b that are not ordered by hb. We use the fact that opsemsets are receptive and closed under sb-prefix to build another candidate pre-execution of P where a and b are still hb-unordered, but for which we can prove it is a pre-execution (not necessarily with the same observable behaviour). From this we deduce that P is not data-race free and ignore these cases.

By construction the pre-execution (O, W) has the same observable behaviour as (O', W'); we conclude by showing that (O', W') can not have undefined behaviours that (O, W) does not have.

The cmmtest tool Building on the theory of the previous section, we designed and implemented a bug-hunting tool called `cmmtest`. The tool performs random testing of C and C++ compilers, implementing a variant of Eide and Regehr's *access summary testing* [35]. A test case is any well-defined, sequential C program; for each test case, `cmmtest`:

1. compiles the program using the compiler and compiler optimisations that are being tested;
2. runs the compiled program in an instrumented execution environment that logs all memory accesses to global variables and synchronisations;
3. compares the recorded trace with a reference trace for the same program, checking if the recorded trace can be obtained from the reference trace by valid eliminations, reorderings and introductions.

We tested the latest svn version of the 4.7 and 4.8 branches of the `gcc` compiler with the `cmmtest` tool. We reported several concurrency bugs (including bugs no. 52558, 54149, 54900, and 54906 in the `gcc` bugzilla), which have all been promptly fixed by the `gcc` developers. In one case the bug report highlights an obscure corner case of the `gcc` optimiser, as shown by a discussion on the `gcc-patches` mailing list[2]. In all cases the bugs were wrongly introduced

[2] `http://gcc.gnu.org/ml/gcc-patches/2012-10/msg01411.html`

writes, speculated by the LIM or IFCVT (if-conversion) phases, similar to the example in Figure 2.1. These bugs do not only break the C11/C++11 memory model, but also the Posix DRF-guarantee which is assumed by most concurrent software written in C and C++. The corresponding patches are activated via the `---param allow-store-data-races=0` flag, which will eventually become default standard for `-std=c11` or `-std=c++11` flags. All these are *silent wrong-code bugs* for which the compiler issues no warning.

The matching algorithm of the `cmmtest` tool can be easily modified to check for compiler invariants rather than for the most permissive sound optimisations, and makes possible to catch unexpected compiler behaviours. For instance, in the current phase of development, `gcc` forbids all reorderings of a memory access with an atomic one. Once we baked this invariant into `cmmtest`, in less than two hours of testing on an 8-core machine we found a sample program that breaks the above invariant.

We stress that none of these could have been found using the existing compiler testing methods.

2.2 Semantic engineering to reuse thread-local reasoning

Problem Reasoning on sequential code is hard enough. When we move from sequential to concurrent computation is it possible to reuse existing large developments relative to the sequential case?

In this section we go through two case studies in which semantic engineering enables the reuse in a concurrent setting of large parts of existing proofs about sequential computation.

2.2.1 Reusing CompCert's sequential correctness proofs in CompCertTSO

In the sequential setting, verified compilation has recently been shown to be feasible by Leroy et al.'s CompCert [59, 60]. CompCert 1.5, our starting point, is a verified compiler from a sequential C-like language, Clight, to PowerPC and ARM assembly language [58][3].

We consider verified compilation in the setting of concurrent programs with a realistic relaxed memory model, and build CompCertTSO, a verified compiler for relaxed memory concurrency on the x86 architecture, derived from CompCert 1.5. It compiles a C-like language called ClightTSO, derived from CompCert's Clight [20], that exposes the x86 hardware load and store operations and synchronisation primitives to the programmer, so ClightTSO loads and stores inherit the hardware relaxed-memory TSO behaviour.

Although the semantic design of ClightTSO turns out to involve a surprisingly delicate interplay between the relaxed memory model, the behaviour of block allocation and free, and the behaviour of pointer equality, in this section we focus on the *semantic engineering* that made CompCertTSO possible. Relaxed memory models are complex in themselves, and a verified compiler such as CompCert is complex even in the sequential case; to make verified compilation for a concurrent relaxed-memory language feasible we have to pay great attention to structuring the semantics of the source and target languages, and the compiler and any correctness proof, to separate concerns and re-use as much as possible. As we shall

[3]More recent CompCert versions start from a higher-level *CompCert C* language and add an x86 target.

see, thread-wise reasoning will allow us to reuse CompCert's proofs for all compiler phases that do not modify memory accesses.

Correctness statement The first question is the form of the correctness theorems that we would like the compiler to generate. We confine our attention to the behaviour of whole programs, leaving a compositional understanding of compiler correctness for relaxed-memory concurrency (e.g. as in the work of Benton and Hur for sequential programs [19]) as a problem for future work. The semantics of ClightTSO and x86-TSO programs will be labelled transition systems (LTS) with internal τ transitions and with visible events for call and return of external functions (e.g. OS I/O primitives), program exit, and semantic failure:

$$\mathit{event},\ \mathit{ev} ::= \texttt{call}\ \mathit{id}\ \mathit{vs} \mid \texttt{return}\ \mathit{typ}\ v \mid \texttt{exit}\ n \mid \texttt{fail}$$

We split external I/O into call and return transitions so that blocking OS calls can be correctly modelled.

Now, how should the source and target LTS be related? As usual for implementations of concurrent languages, we cannot expect them to be equivalent in any sense, as the implementation may resolve some of the source-language nondeterminism (c.f. [86] for earlier discussion of the correctness of concurrent language implementations). For example, in our implementation, stack frames will be deterministically stack-allocated and the pointers in the block-reuse example above will always be equal. Hence, the most we should expect is that if the compiled program has some observable behaviour then that behaviour is admitted by the source semantics — an inclusion of observable behaviour.

This must be refined further: compiled behaviour that arises from an erroneous source program need not be admitted in the source semantics (e.g. if a program mutates a return address on its stack, or tries to apply a non-function). The compiled program should only diverge, indicated by an infinite trace of τ labels, if the source program can. Moreover, without a quantitative semantics, we have to assume that the target language can run out of memory at any time. We capture all this with the following definition of LTS *trace*.

Traces *tr* are either infinite sequences of non-`fail` visible events or finite sequences of non-`fail` visible events ending with one of the following three markers: end (designating successful termination), inftau (designating an infinite execution that eventually stops performing any visible events), or oom (designating an execution that ends because it runs out of memory). The traces of a program p are given as follows:

$$\begin{aligned}
\mathsf{traces}(p, \mathit{args}) \stackrel{\text{def}}{=}\ & \{\ell \cdot \mathsf{end} \mid \exists s \in \mathsf{init}(p, \mathit{args}).\ \exists s'.\ s \stackrel{\ell}{\Rightarrow} s' \not\rightarrow\} \\
\cup\ & \{\ell \cdot \mathsf{inftau} \mid \exists s \in \mathsf{init}(p, \mathit{args}).\ \exists s'.\ s \stackrel{\ell}{\Rightarrow} s' \stackrel{\tau}{\rightarrow}^{\omega}\} \\
\cup\ & \{\ell \cdot \mathit{tr} \mid \exists s \in \mathsf{init}(p, \mathit{args}).\ \exists s'.\ s \stackrel{\ell}{\Rightarrow} s' \xrightarrow{\texttt{fail}}\} \\
\cup\ & \{\ell \cdot \mathsf{oom} \mid \exists s \in \mathsf{init}(p, \mathit{args}).\ \exists s'.\ s \stackrel{\ell}{\Rightarrow} s'\} \\
\cup\ & \{l \mid \exists s \in \mathsf{init}(p, \mathit{args}).\ s \stackrel{l}{\Rightarrow} \text{ and } l \text{ is infinite}\}
\end{aligned}$$

Here $\mathsf{init}(p, \mathit{args})$ denotes the initial states for a program p when called with command-line arguments *args*; for a finite sequence ℓ of non-`fail` visible events, we define $s \stackrel{\ell}{\Rightarrow} s'$ to hold whenever s can do the sequence ℓ of events, possibly interleaved with a finite number of τ-events, and end in state s'; and for a finite or infinite sequence l of non-`fail` visible events,

we define $s \stackrel{l}{\Rightarrow}$ to hold whenever s can do the sequence l of events, possibly interleaved with τ-events.

We treat failed computations as having arbitrary behaviour after their failure point, whereas we allow the program to run out of memory at any point during its execution. This perhaps-counter-intuitive semantics of oom is needed to express a correctness statement guaranteeing nothing about computations that run out of memory.

Our top-level correctness statement for a compiler $\mathsf{compile}$ from ClightTSO to x86-TSO, modelled as a partial function, will then be a trace inclusion for programs for which compilation succeeds, of the form

$$\forall p, \mathit{args}.\ \mathsf{defined}(\mathsf{compile}(p)) \implies \mathsf{traces}_{\mathsf{x86\text{-}TSO}}(\mathsf{compile}(p), \mathit{args}) \subseteq \mathsf{traces}_{\mathsf{ClightTSO}}(p, \mathit{args})$$

where the functions $\mathsf{traces}_{\mathsf{x86\text{-}TSO}}$ and $\mathsf{traces}_{\mathsf{ClightTSO}}$ build all the traces of a given program when run with the given arguments, according to the assembly or source language semantics.

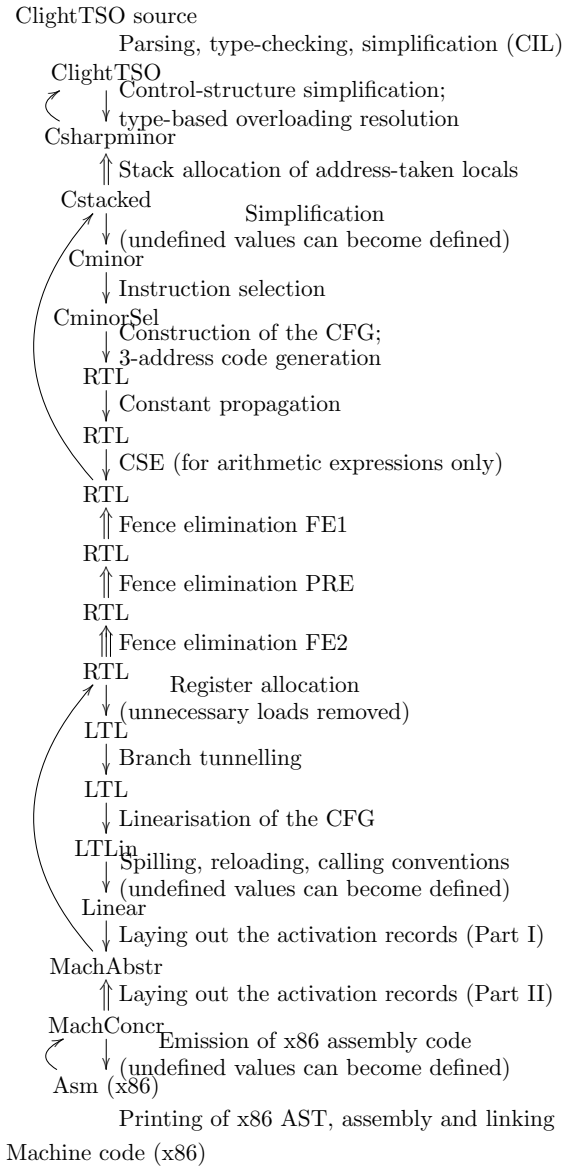
The CompCert 1.5 proof strategy ClightTSO is an extension of sequential Clight, and its compiler has to deal with everything that a Clight compiler does, except for any optimisations that become unsound in the concurrent setting. We therefore arrange our semantic definitions and proof structure to re-use as much as possible of the CompCert development for sequential Clight, isolating the parts where relaxed-memory concurrency plays a key role.

Our starting point was CompCert 1.5 which is subdivided into 13 compiler phases, each of which builds a semantic preservation proof between semantically defined intermediate languages. The overall strategy is to prove trace inclusions by establishing simulation results — more particularly, to build some kind of "downward" simulation for each phase, showing that transitions of a source program for the phase can be matched by transitions of the compiled target program; these can be composed together and combined with determinacy for the target language (there PowerPC or ARM assembly) to give an upward simulation for a complete compilation, showing that any behaviour of a compiled program is allowed by the source program semantics.[4] Downward simulations are generally easier to establish than upward simulations because compiler phases tend to introduce intermediate states; a downward simulation proof does not have to characterise and relate these.

As we shall see, this strategy cannot be used directly for compilation of concurrent ClightTSO to x86, but much can be adapted.

Decomposing the proof by compiler phases Our compiler is divided into similar (but not identical) phases to CompCert 1.5, illustrated in Figure 2.4. For each phase, we define the semantics of a whole program to be an LTS as above, and inclusion of the above notion of traces also serves as the correctness criterion for each of our phases. The individual correctness results can be composed simply by transitivity of set inclusion.

[4]Terminology: in CompCert "forward simulation" and "backward simulation" refer to the direction of the simulation with respect to the compiler phases, thinking of compilation as "forwards". This clashes with another standard usage in which "forward" and "backward" refer to the direction of transitions. In this paper we need to discuss both, so we use "downwards" (and conversely "upwards") to refer to the direction



Our proof structure is indicated by single arrows for threadwise downward simulations (lifted to upward trace inclusions, shown with upwards single arrows, by Theorems 2.2.5, 2.2.7 and 2.2.8); straight double (resp. triple) arrows for direct proofs of whole-system upward simulations (resp. upward weak-tau simulation).

ClightTSO and Csharpminor perform a stack allocation for each individual variable in the program and assume an infinite memory, whereas the languages below have only finite memory. From Cstacked to MachAbstr a stack allocation occurs for each non-empty stack frame (that is, almost every function call), whereas in MachConcr and Asm a stack allocation occurs only when a thread is created.

Figure 2.4: CompCertTSO phases

Labellisation and threadwise proof In our concurrent setting the languages are not deterministic, so the CompCert approach to building upward simulations is not applicable. However, for most of the phases we can re-use the CompCert proof, more-or-less adapted, to give downward simulation results for the behaviour of a single thread in isolation — and we can make our semantics deterministic for such. We therefore 'labellise' the semantics for each level (source, target, and each intermediate language). Instead of defining transitions

$$(s, m_{\mathrm{SC}}) \longrightarrow (s', m'_{\mathrm{SC}})$$

over configurations that combine a single-threaded program state s and a (sequentially consistent) memory m_{SC} (as most sequential language semantic definitions, including CompCert 1.5, do), we define the semantics of a single thread (split apart from the memory) as a transition system:

$$s \xrightarrow{te} s'$$

(together with extra structure for thread creation) where a thread event te is either an external event, as above, an interaction with memory me, an internal τ action, the start or exit of the thread, or an out-of-memory error `oom`:

$$\mathit{thread\_event},\ te ::= \quad \texttt{ext}\ ev \mid \texttt{mem}\ me \mid \tau \mid \texttt{start}\ \mathit{opt\_tid}\ p\ vs \mid \texttt{exit} \mid \texttt{oom}$$

The whole-system semantics of each level is a parallel composition roughly of the form

$$s_1 \mid \ldots \mid s_n \mid m_{\mathrm{TSO}}$$

of the thread states s_i and a TSO memory m_{TSO}. The threads interact with the TSO memory by synchronising on various events: reads or writes of a pointer p with a value v of a specified *memory_chunk* size, allocations and frees of a memory block at a pointer p, various error cases, and thread creation.[5] Analogously to the events of the previous section, these transitions are in the style of the 'early' transition system for value-passing CCS [66]: a thread doing a memory read will have a transition for each possible value of the right type. For example, here is the ClightTSO rule for dereferencing a pointer:

$$\frac{\begin{array}{l}\texttt{access\_mode}\ ty' = \texttt{By\_value}\ c \\ \mathit{typ} = \texttt{type\_of\_chunk}\ c \\ \texttt{Val.has\_type}\ v\ \mathit{typ}\end{array}}{p \cdot \left[\texttt{*\_}^{ty'}\right] \cdot \kappa_{\mathrm{e}}\,|_{\rho} \xrightarrow{\texttt{mem (read}\ p\ c\ v\texttt{)}} v \cdot \kappa_{\mathrm{e}}\,|_{\rho}} \quad \textsc{LoadByValue}$$

The conclusion has a start state with a pointer value p in an expression continuation $\left[\texttt{*\_}^{ty'}\right] \cdot \kappa_{\mathrm{e}}$ headed by a dereference at a ClightTSO type ty'. The first premise finds the access mode of that type: here it must be accessed by value and has a chunk c (specifying int/float, size, and signedness). The second premise collapses this onto an internal type *typ* (just int/float, because internal values do not record their size or signedness). The third premise

of compilation, reserving "forwards" and "backwards" for the direction of transitions. (Notwithstanding this, the CompCertTSO sources retain the CompCert usage.)

[5] For the purposes of this section the exact definition of the semantics of the TSO memory is irrelevant, details can be found in [74] and [85].

allows an arbitrary value v of type *typ*. Then the conclusion has a transition labelled with a memory read, at pointer p, of that value v, as a chunk c, to a state with v in the remaining continuation. (There is a further subtlety here. One might think that the rule should also check that v represents a value of type ty', not just that it has internal type *typ*. That check could be added here, but in fact we have it in the TSO machine. The premises do suffice to ensure a receptiveness property, which is what we really need of the thread semantics.)

External events of the threads (and of the TSO machine) are exposed directly as the whole-system behaviour.

This conceptually simple change to a labellised semantics separates concerns: compiler phases that do not substantially affect the memory accesses of the program can be proved correct per-thread and those results lifted to the whole system by a general result below, leaving only the two remaining main phases and three fence optimisation phases that require proofs that really involve the TSO machine.

More in detail, for the phases that do not substantially change memory accesses, we establish whole-system trace inclusions from threadwise downward simulations in three steps. First, we observe that a downward simulation from a receptive language to a determinate language implies the existence of upward simulation and use this to obtain threadwise upward simulation. Then we lift the threadwise upward simulation to a whole-system upward simulation. Finally, we establish trace inclusion from the whole-system upward simulation.

We say that two labels are of the same kind, written $te \asymp te'$ if they only differ in input values. In our case, $te \asymp te'$ if (i) te and te' are reads from the same memory location (but not necessarily with the same value), or (ii) te and te' are external returns, or (iii) $te = te'$.

Definition 2.2.1 *A thread LTS is* receptive *if* $s \xrightarrow{te} t$ *and* $te' \asymp te$ *implies* $\exists t'.\ s \xrightarrow{te'} t'$.

Definition 2.2.2 *A thread LTS is* determinate *if* $s \xrightarrow{te} t$ *and* $s \xrightarrow{te'} t'$ *implies* $te \asymp te'$ *and, moreover, if* $te = te'$, *then* $t = t'$.

Definition 2.2.3 *A relation R between the states of two thread LTSs S and T is a* threadwise downward simulation *if there is a well-founded order $<$ on the states of S such that if given any $s, s' \in S$, $t \in T$ and label te, whenever $s \xrightarrow{te} s'$ and $s\ R\ t$, then either*

1. $te = \texttt{fail}$, *or*
2. $\exists t'.\ t \xrightarrow{\tau}{}^{*} \xrightarrow{te} \xrightarrow{\tau}{}^{*} t' \wedge s'\ R\ t'$, *or*
3. $te = \tau \wedge s'\ R\ t \wedge s' < s$.

Definition 2.2.4 *A relation R is a* threadwise upward simulation *if there is a well-founded order $<$ on T such that whenever $t \xrightarrow{te} t'$ and $s\ R\ t$, then either*

1. $\exists s'.\ s \xrightarrow{\tau}{}^{*} \xrightarrow{te} s' \wedge s'\ R\ t'$, *or*
2. $\exists s'.\ s \xrightarrow{\texttt{fail}} s'$, *or*
3. $te = \tau \wedge s\ R\ t' \wedge t' < t$.

Moreover, if $t \not\rightarrow$ *(*t *is stuck) and* $s\ R\ t$*, then* $s \not\rightarrow$ *or* $\exists s'.\ s \xrightarrow{\texttt{fail}} s'$*.*

Note the subtle asymmetry in handling errors: if a source state signals an error or gets stuck, both the upward simulation and downward simulation hold. In contrast, the target states' errors must be reflected in the source to make the upward simulation hold. This is necessary to allow compilers to eliminate errors but not to introduce them.

Theorem 2.2.5 *If* R *is a threadwise downward simulation from* S *to* T*,* S *is receptive, and* T *is determinate, then there is a threadwise upward simulation that contains* R*.*

Eliding details of initialisation and assumptions on global environments, we have:

Definition 2.2.6 *A relation* $R : \mathsf{States}(S) \times \mathsf{States}(T)$*, equipped with a well-founded order* $<$ *on* $\mathsf{States}(T)$*, is a* measured upward simulation *if, whenever* $s\ R\ t$ *and* $t \xrightarrow{ev} t'$*, then either*

1. $\exists s'.\ s \xrightarrow{\tau}{}^{*}\ s' \xrightarrow{\texttt{fail}}$ *(*s *can reach a semantic error), or*
2. $\exists s'.\ s \xrightarrow{\tau}{}^{*} \xrightarrow{ev} s' \wedge s'\ R\ t'$ *(*s *can do a matching step), or*
3. $ev = \tau \wedge t' < t \wedge s\ R\ t'$ *(*t *stuttered, with a decreasing measure).*

Theorem 2.2.7 *A threadwise upward simulation can be lifted to a whole-system measured upward simulation, for the composition of the threads with the TSO machine.*

Theorem 2.2.8 *A whole-system upward simulation implies trace inclusion.*

A concluding remark. This case-study additionally shows how working with full-scale systems requires to deal with swarms of cases: compare for instance the defining clauses in the definition of traces, the condition to establish a simulation, or the LoadByValue transition rule, with the analgous definitions for similar results on idealised process calculi. To make the situation worse, it is often the case that general theorems require to be specialised to take into account peculiar cases. For instance, to establish correctness of compiler phases that remove dead variable loads and concretise undefined values, we have also proved variants of Theorems 2.2.5 and 2.2.7 for suitably modified Definitions 2.2.3 and 2.2.4. However today advances in formal methods and proof assistants make these difficulties tractable, and research should not oversimply the complexity of modern systems.

2.2.2 Thread-wise reasoning via oracular semantics

In [46] we designed and proved sound a concurrent separation logic for an extension of Cminor with threads and locks, called Concurrent Cminor. The semantics of Concurrent Clight is given by a whole-system transition relation of the form:

$$(s_1, \ldots, s_n, m_{\mathrm{SC}}) \longrightarrow (s'_1, \ldots, s'_n, m'_{\mathrm{SC}})$$

where the s_i denote the local state of thread i and m_{SC} is the (sequentially consistent) shared memory. However, a triple $\{P\}c\{Q\}$ in separation logic (or even a compiler), considers a

single thread at a time and we need a deterministic sequential semantics that knows how to handle concurrent communications. In Concurrent Cminor only well-synchronised programs are defined, so most semantic rules of single-thread computation are unaffected by other threads.

A general technique to build the desired semantics is to rely on an *oracular machine*, where the oracle, denoted by Ω, contains a schedule, represented as a list of thread indexes to be run at each context switch and denoted by $\mho$, and a list of the threads with their local state $\overline{s}$. In other terms the oracle knows the concurrent environment and the schedule of the threads so that it can deterministically simulate the execution of the concurrent environment up-to the point when control is returned to the thread under consideration (if ever). The reductions thus have the form

$$(\Omega, s_i, m_{\mathrm{SC}}) \longmapsto (\Omega', s_i', m_{\mathrm{SC}}')$$

The semantics of single-thread computation is straightforward and does not affect the oracle; when the oracular machine gets to a concurrent instruction instead it builds the concurrent machine $(\overline{s}, s_i, m_{\mathrm{SC}})$ and reduces it following the $\longrightarrow$ transition, at each step picking the thread whose index is specified by the schedule. If the schedule returns the control to thread i, for instance in state $(\overline{s}', s_i, m_{\mathrm{SC}}')$ with remaining schedule $\mho'$, then the oracular machine returns the oracular state $((\overline{s}', \mho'), s_i, m_{\mathrm{SC}}')$ and the thread under consideration now knows how the global memory has been modified by the environment.

Classical reasoning is unavoidable within the oracular semantics: determining if control will return to a given thread reduces to the halting problem. The nonconstructivity of this operational semantics is not a bug: the oracular semantics is not an interpreter but a specification for correctness proofs of compilers and program logics. Proofs still have to quantify over all oracles: the oracular machine does not simplify the contextual reasoning but the oracular step is used to keep "unimportant" details of the concurrent machine from interfering with proofs about the sequential language (which can then be reused), and as such can be seen as another example of how *semantic engineering* can make large proofs tractable.

2.3 The way forward

In the previous sections we have seen three cases where it is possible to perform thread-wise reasoning, exploiting either the relationship between the events performed by the to-be-proved equivalent system or a global knowledge of the environment.

However, a general technique for compositional reasoning for shared memory concurrent systems is still missing. The previously unpublished discussion that follow explores the complexity of the general case and makes a case for the way forward.

Consider this simple model of shared memory concurrent systems:

- *shared memory locations* are denoted by x,y,z,...
- *values*, denoted by v range over some finite subset of the naturals, e.g. {0,1}

- *actions*, ranged over by a, comprise internal reductions (denoted τ), writing value v to the shared memory location x (denoted by $\mathrm{W}\,x\,v$), and reading value v from the shared memory location x (denoted by $\mathrm{R}\,x\,v$).
- *threads*, ranged over by P,Q are LTSs over actions. We denote transitions as $P_1 \xrightarrow{a} P_2$. For convenience we assume threads have no infinite paths, but we assume at least that they are receptive: for all reachable P', if $P' \xrightarrow{\mathrm{R}\,x\,v} P''$ then forall values v', there exists P''' such that $P' \xrightarrow{\mathrm{R}\,x\,v'} P'''$. Parallel composition of threads $P|Q$ is their free interleaving:

$$\frac{P \xrightarrow{a} P'}{P|Q \xrightarrow{a} P'|Q} \qquad \frac{Q \xrightarrow{a} Q'}{P|Q \xrightarrow{a} P|Q'}$$

- *store* (denoted by s) is a function from locations to values. In the default initial store, denoted s_0, all locations map to a given value, e.g. 0.
- *composition* of a process with a store, denoted $P \parallel s$ is an unlabelled transition system

$$\frac{P \xrightarrow{\mathrm{R}\,x\,v} P' \quad s(x) = v}{P \parallel s \to P' \parallel s} \qquad \frac{P \xrightarrow{Wxv} P'}{P \parallel s \to P' \parallel (s + (x \mapsto v))} \qquad \frac{P \xrightarrow{\tau} P'}{P \parallel s \to P' \parallel s}$$

- the *final states* of a thread P, denoted $\mathrm{fs}(P)$, are defined as

$$\mathrm{fs}(P) = \{s \mid P \parallel s_0 \to^* P \parallel s \not\to\}$$

  (we are in linear time and we ignore nonterminating executions).

We define *observational congruence* based on those as:

equivalence $P \cong P'$ iff forall Q. $\mathrm{fs}(P|Q) \;=\; \mathrm{fs}(P'|Q)$

preorder $P <_c P'$ iff forall Q. $\mathrm{fs}(P|Q) \;\subseteq\; \mathrm{fs}(P'|Q)$

Is it possible to give a characterisation of $P \cong P'$ that does not involve the quantification on the arbitrary context Q? In the previous sections we have seen cases in which this possible, for instance when P' is obtained from P via a compiler optimisation. But what about the general case when P and P' are a priori unrelated? Is it possible to perform contextual reasoning on concurrent shared memory programs? As far as we know, this fundamental, and apparently innocent, question is still open. Below we list some examples of equivalent processes.

- It holds that

$$\mathrm{W}\,x\,1.\mathrm{W}\,x\,1 + \mathrm{W}\,x\,1 \;\cong\; \mathrm{W}\,x\,1.\mathrm{W}\,x\,1 \qquad \text{(write stuttering)}$$

  but

$$\mathrm{W}\,x\,1.\mathrm{W}\,y\,1.\mathrm{W}\,x\,1.\mathrm{W}\,y\,1 + \mathrm{W}\,x\,1.\mathrm{W}\,y\,1 \;\ncong\; \mathrm{W}\,x\,1.\mathrm{W}\,y\,1.\mathrm{W}\,x\,1.\mathrm{W}\,y\,1$$

- It holds that

$$\mathrm{R}\,x\,0.\mathrm{W}\,z\,1 + \mathrm{R}\,x\,1.\mathrm{W}\,z\,1 \;\cong\; \mathrm{W}\,z\,1 \qquad \text{(irrelevant reads)}$$

  and

$$\mathrm{R}\,x\,0.(\mathrm{R}\,y\,0.\mathrm{W}\,z\,1 + \mathrm{R}\,y\,1.\mathrm{W}\,z\,1) + \mathrm{R}\,x\,1.(\mathrm{R}\,y\,0.\mathrm{W}\,z\,1 + \mathrm{R}\,y\,1.\mathrm{W}\,z\,1) \;\cong\; \mathrm{W}\,z\,1$$

- It holds that

$$\mathrm{W}\,x\,1 + \mathrm{W}\,x\,2 \;\cong\; \mathrm{R}\,y\,0.(\mathrm{W}\,x\,1 + \mathrm{W}\,x\,2) + \mathrm{R}\,y\,1.\mathrm{W}\,x\,1$$

  but are these processes $P1 = \mathrm{W}\,y\,1$ and $P2 = \mathrm{R}\,y\,0.\mathrm{W}\,y\,1 + \mathrm{R}\,y\,1$ equivalent?

- In some cases equivalences are true for boring reasons, here are some general laws:

$$P <_c P + Q \qquad P <_c P' \;\Rightarrow\; P + Q <_c P' + Q \qquad P <_c Q \;\Rightarrow\; P + Q \cong Q$$

  For the write-only case the equation

$$\mathrm{W}\,x\,1 <_c \mathrm{W}\,x\,1.\mathrm{W}\,x\,1$$

  holds for an interesting reason: the rhs can always choose to do its writes close together in order to match any behaviour of the lhs. In turn this implies write stuttering by the third general rule above.

- It is possible to remove writes if they can write any value without introducing new behaviour, i.e.,

$$P <_c \mathrm{W}\,y\,0.P + \mathrm{W}\,y\,1.P \quad \text{provided that } 0\ 1 \text{ are the only possible values.}$$

  This is in the same direction as the "interesting" direction of irrelevant reads above, i.e. $\mathrm{W}\,z <_c \mathrm{R}\,x\,0.\mathrm{W}\,z + \mathrm{R}\,x\,1.\mathrm{W}\,z$, and for roughly the same reason: it we consider an execution of $P|C$ then we pay attention to what values y (or x respectively) has, and build a matching trace of $Q|C$ in which we write (or read, respectively) exactly those values, thereby having no effect.

- Let traceset of WRITE-CHAOS be the set of all possible finite (i.e., terminating) sequences of writes. Then it would be the case that

$$\mathrm{W}\,x\,0.\mathsf{WRITE\text{-}CHAOS} \;\cong\; \mathrm{W}\,y\,0.\mathsf{WRITE\text{-}CHAOS}$$

- The threads $P = \mathrm{R}\,x\,0.(\mathrm{W}\,z\,0 + \mathrm{W}\,z\,1) + \mathrm{R}\,x\,1.\mathrm{W}\,z\,0$ and $Q = \mathrm{W}\,z\,0 + \mathrm{R}\,x\,1.\mathrm{W}\,z\,0 + \mathrm{R}\,x\,0.\mathrm{W}\,z\,1$ should be equated by $\cong$ (I conjecture).

These examples show that the equational theory of shared memory processes is non-trivial. As far as we know, the only published attempt at characterising it is given by Brookes in [25], which shows that it is possible to build a normal form by saturating with respect two relations he calls *stuttering* and *mumbling*, and that equivalence coincides with equality of normal forms. However Brookes setup is not satisfactory: Brookes has assignments (including their expressions) that evaluate atomically, so the (s_i, s'_i) parts of his traces correspond to reads of (part or possibly all of) the shared state s_i together with an atomic update giving the new shared state s'_i. This gives the observed the power to atomically read and update the whole state, which is unrealistic.

Our attempt at giving a characterisation of process congruence while considering reads in isolation resulted in the collection of examples above and not much else, but this is a fundamental problem and it is surprising that it does not have a satisfactory solution after 25 years of research in concurrency theory.

Chapter 3

Tools for the working semanticist

In which we design tools to write better programming language papers.

Problem Writing a precise semantic definition of a full-scale programming language is a challenging task that has been done only rarely, despite the many potential benefits. Indeed, Standard ML remains, 19 years after publication, the shining example of a language that is defined precisely and is at all widely used [67]. The recent R^{6}RS Scheme standard [98] contains a (non-normative) operational semantics for a large part of the language, but even languages such as Haskell [76] and OCaml [61], though designed by programming language researchers and in large part based on mathematical papers, rely on prose descriptions.

Precise semantic definitions are rare for several reasons, but one important reason is that the *metalanguages* that are available for expressing semantic definitions are not designed for this application, making it much harder than necessary to work with large definitions. There are two main choices for a metalanguage:

(1) Informal mathematics, expressed in LaTeX (by far the most common option).

(2) Formalised mathematics, in the language of a proof assistant such as Coq, HOL, Isabelle/HOL, or Twelf [33, 47, 50, 102].

For a small calculus either can be used without much difficulty. A full language definition, however, might easily be 100 pages or 10 000 lines. At this scale the syntactic overhead of LaTeX markup becomes very significant, getting in the way of simply reading and writing the definition source. The absence of automatic checking of sanity properties becomes a severe problem — in our experience with the Acute language [87, 89], just keeping a large definition internally syntactically consistent during development is hard, and informal proof becomes quite unreliable, as highlighted by the POPLmark challenge [15]. Further, there is no support for relating the definition to an implementation, either for generating parts of an implementation, or for testing conformance. Accidental errors are almost inescapable [52, 78].

Proof assistants help with automatic checking, but come with their own problems. The sources of definitions are still cluttered with syntactic noise, non-trivial encodings are often needed (e.g. to deal with subgrammars and binding, and to work around limitations of the available polymorphism and inductive definition support), and facilities for parsing and pretty printing terms of the source language are limited. Typesetting of definitions is supported only partially and only in some proof assistants, so one may have the problem of

maintaining machine-readable and human-readable versions of the specification, and keeping them in sync. Moreover, each proof assistant has its own (steep) learning curve, the community is partitioned into schools (few people are fluent in more than one), and one has to commit to a particular proof assistant from the outset of a project.

A more subtle consequence of the limitations of the available metalanguages is that they obstruct re-use of definitions across the community, even of small calculi. Research groups each have their own private LaTeX macros and idioms — to build on a published calculus, one would typically re-typeset it (possibly introducing minor hopefully-inessential changes in the process). Proof assistant definitions are more often made available (e.g. in the Archive of Formal Proofs [54]), but are specific to a single proof assistant. Both styles of definition make it hard to compose semantics in a modular way, from fragments.

The Dream We would like to have a metalanguage that is designed for the working semanticist, supporting common notations that have been developed over the years. In an email or working note one might write grammars for languages with complex binding structures, for example

```
t ::=
  | let p = t in t'          bind binders(p) in t'
p ::=
  | x                        binders = x
  | { l1=p1,...,ln=pn }      binders = binders(p1 ... pn)
```

and informal semantic rules, for example as below.

```
G |- t1:T1  ... G |- tn:Tn
-----------------------------------------
G |- {l1=t1,...,ln=tn} : {l1:T1,...,ln:Tn}
```

These are intuitively clear, concise, and easy to read and edit. Sadly, they lack both the precision of proof assistant definitions and the production-quality typesetting of LaTeX. It turns out that only a modicum of information need be added to make them precise, and to automatically *compile* them to both targets.

3.1 Metalanguage design

Fig. 3.1 reports a complete Ott source file for an untyped CBV lambda calculus, including the information required to generate proof assistant definitions in Coq, HOL and Isabelle, OCaml boilerplate, and LaTeX. The typeset LaTeX is shown in Fig. 3.2. This is a very small example, sufficing to illustrate the key aspects of our metalanguage design. However it does not present all the problems of dealing with a full language, which is our main motivation. We comment on those as we go, and invite the reader to imagine the development for their favourite programming language or calculus in parallel.

```
metavar var, x ::=   {{ com  term variable }}
{{ isa string}} {{ coq nat}} {{ hol string}} {{ coq-equality }}
{{ ocaml int}} {{ lex alphanum}} {{ tex \mathit{[[var]]} }}

grammar
term, t :: 't_' ::=                          {{ com term    }}
  | x            ::  :: Var                    {{ com variable}}
  | \ x . t      ::  :: Lam (+ bind x in t +) {{ com lambda  }}
  | t t'         ::  :: App                    {{ com app     }}
  | ( t )        :: S:: Paren                  {{ icho [[t]]  }}
  | { t / x } t' :: M:: Tsub
                       {{ icho (tsubst_t [[t]] [[x]] [[t']])}}

val, v :: 'v_' ::=                           {{ com value   }}
  | \ x . t      ::  :: Lam                    {{ com lambda  }}

terminals :: 'terminals_' ::=
  | \            ::  :: lambda  {{ tex \lambda }}
  | -->          ::  :: red     {{ tex \longrightarrow }}

subrules
  val <:: term

substitutions
  single t x :: tsubst

defns
Jop :: '' ::=

 defn
 t1 --> t2 :: ::reduce::'' {{ com [[t1]] reduces to [[t2]]}} by


    -------------------------  :: ax_app
    (\x.t1) v2 -->  {v2/x}t1

    t1 --> t1'
    ------------- :: ctx_app_fun
    t1 t --> t1' t

    t2 --> t2'
    ------------- :: ctx_app_arg
    v t2 --> v t2'
```

Figure 3.1: A small `ott` source file, for an untyped CBV lambda calculus, with data for Coq, HOL, Isabelle, LaTeX, and OCaml.

Core At first ignore the data within `{{ }}` and `(+ +)`, and the `terminals` block. At the top of the figure, the **`metavar`** declaration introduces *metavariables* `var` (with synonym `x`), for term variables. The following **`grammar`** introduces grammars for terms, with *nonterminal root* `term` (with synonym `t`), and for values `val` (with wynonym) `v`.

```
term, t :: 't_' ::=
  | x             ::   :: Var
  | \ x . t       ::   :: Lam
  | t t'          ::   :: App
  | ( t )         :: M :: Paren
  | { t / x } t' :: M :: Tsub

val, v :: 'v_' ::=
  | \ x . t       ::   :: Lam
```

This specifies the concrete syntax of object-language terms, the abstract syntax representations for proof-assistant mathematics, and the syntax of symbolic terms to be used in semantic rules. The *terminals* of the grammar (`\ . ( ) { } / -->`) are inferred, as those tokens that cannot be lexed as metavariables or nonterminals, avoiding the need to specify them explicitly.

Turn now to the **`defns`** block at the bottom of the figure. This introduces a mutually recursive collection of judgments, here just a single judgement `t1 --> t2` for the reduction relation, defined by three rules. Consider the innocent-looking CBV beta rule:

```
-------------------------  :: ax_app
(\x.t1) v2 -->  {v2/x}t1
```

Here the conclusion is a term of the syntactic form of the judgement being defined, `t1 --> t2`. Its two subterms `(\x.t1) v2` and `{v2/x}t1` are *symbolic* terms for the `term` grammar, not concrete terms of the object language. They involve some object-language constructors (instances of the `Lam` and `App` productions of the `t` grammar), just as concrete terms would, but also:

- mention symbolic metavariables (`x`) and nonterminals (`t1` and `v2`), built from the metavariable and nonterminal roots (`x`, `t`, and `v`) by appending structured suffixes — here just numbers;
- depend on a subtype relationship between `val` and `term` (declared by the **`subrules`** `val <:: term`, and checked by the tool) to allow `v2` to appear in a position where a term of type `t` is expected; and
- involve syntax for parentheses and substitution. The concrete syntax for these is given by the `Paren` and `Tsub` productions of the `t` grammar, but these are *metaproductions* (flagged **`M`**), for which we do not want abstract syntax constructors.

The `ax_app` rule does not have any premises, but the other two rules do, e.g.

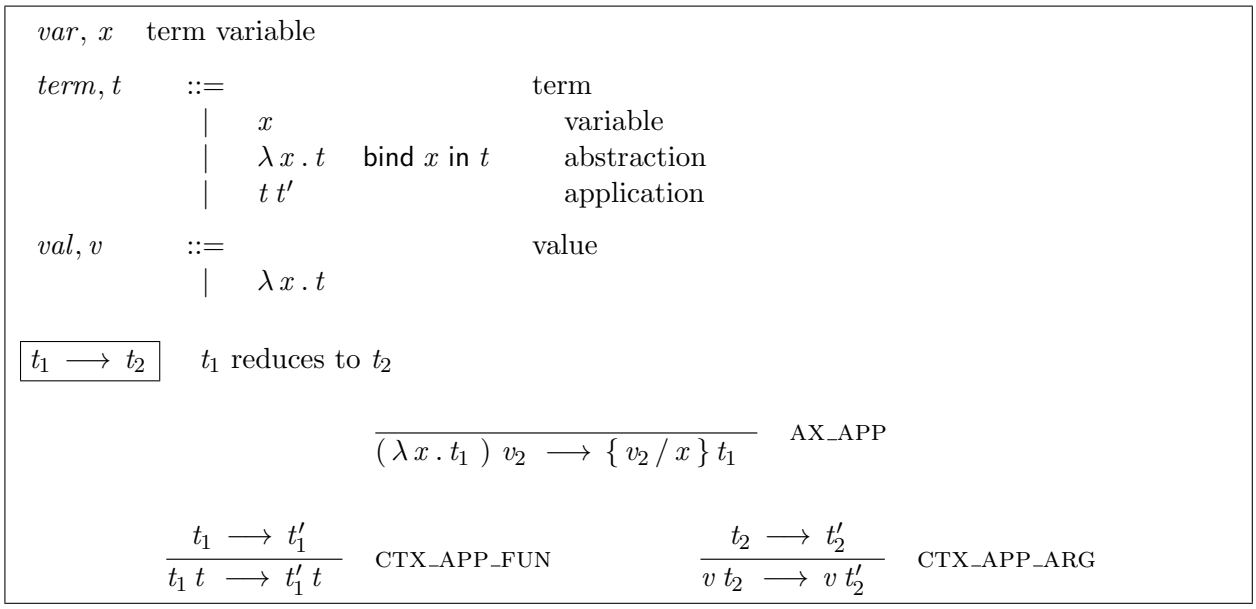


Figure 3.2: LaTeX output generated from the Fig. 3.1 source file

```
t2 --> t2'
-------------- :: ctx_app_arg
v t2 --> v t2'
```

Here the premises are instances of the judgement being defined, but in general they may be symbolic terms of a `formula` grammar that includes all judgement forms by default, but can also contain arbitrary user-defined formula productions, for side-conditions.

This core information is already a well-formed `ott` source file that can be processed by the tool, sanity-checking the definitions, and default typeset output can be generated.

Proof assistant code To generate proof assistant code we first need to specify the proof assistant representations ranged over by metavariables: the **`isa`**, **`coq`** and **`hol`** annotations of the **`metavar`** block specify that the Isabelle, Coq and HOL `string`, `nat` and `string` types be used. For Coq the **`coq-equality`** generates an equality decidability lemma and proof script for the type.

The proof assistant representation of abstract syntax is then generated from the grammar. For a very simple example, the Coq compilation for `term` generates a free type with three constructors:

```
Inductive term : Set :=
   t_Var : var -> term
 | t_Lam : var -> term -> term
 | t_App : term -> term -> term.
```

The general case is rather more complex than this, but here we just note that the metaproductions do not give rise to proof assistant constructors. Instead, the user can specify an arbitrary translation for each.

$$\frac{\begin{array}{l} E \vdash e_1 : t_1 \quad ... \quad E \vdash e_n : t_n \\ E \vdash \mathit{field\_name}_1 : t \to t_1 \quad ... \quad E \vdash \mathit{field\_name}_n : t \to t_n \\ t = (\, t'_1 \,, \, ... \,, \, t'_l \,)\, \mathit{typeconstr\_name} \\ E \vdash \mathit{typeconstr\_name} \;\rhd\; \mathit{typeconstr\_name} : \mathit{kind}\, \{\, \mathit{field\_name}'_1 \,; \, ... \,; \, \mathit{field\_name}'_m \,\} \\ \mathit{field\_name}_1 \, ... \, \mathit{field\_name}_n \; \mathbf{PERMUTES} \; \mathit{field\_name}'_1 \, ... \, \mathit{field\_name}'_m \\ \mathbf{length}\, (\, e_1 \,) \, ... \, (\, e_n \,) \geq 1 \end{array}}{E \vdash \{\, \mathit{field\_name}_1 = e_1 \,; \, ... \,; \, \mathit{field\_name}_n = e_n \,\} : t} \;\; \textsc{JTe\_rc}$$

```
E |- e1 : t1 ... E |- en : tn
E |- field_name1 : t->t1 ... E |- field_namen : t->tn
t = (t1', ..., tl') typeconstr_name
E |- typeconstr_name gives typeconstr_name:kind {field_name1'; ...; field_namem'}
field_name1...field_namen PERMUTES field_name1'...field_namem'
length (e1)...(en)>=1
------------------------------------------------------------------------ :: rc
E |- {field_name1=e1; ...; field_namen=en} : t
```

Figure 3.3: A sample OCaml semantic rule, in LaTeX and `ott` source forms

These translations ('*homs*') give clauses of functions from symbolic terms to the character string of generated proof-assistant code. In this example, the `{{ icho [[t]] }}` hom for the `Paren` production says that `(t)` should be translated into just the translation of `t`, whereas the `{{ icho (tsubst_t [[t]] [[x]] [[t']])}}` hom for `Tsub` says that `{t/x}t'` should be translated into the proof-assistant application of `tsubst_t` to the translations of `t`, `x`, and `t'`. The (admittedly terse) '**icho**' specifies that these translations should be done uniformly for Isabelle, Coq, HOL, and OCaml output, but one can also specify different translations for each.

The `tsubst_t` mentioned in the hom for `Tsub` above is a proof assistant identifier for a function that calculates substitution over terms, automatically generated by the **substitutions** declaration. In the next section we explain what this does, and to the meaning of the binding specification `(+ bind x in t +)` in the `Lam` production.

Homs can also be used to specify proof assistant *types* for nonterminals, in cases where one wants a specific proof assistant type expression rather than a type freely generated from the syntax. More generally, as we shall see, this homomorphism machinery is useful for several different purposes.

Tuned typesetting To fine-tune the generated LaTeX, to produce the output of Fig. 3.2, the user can add various data: (1) the `{{ tex \mathit{[[termvar]]} }}` in the **metavar** declaration, specifying that `termvar`s be typeset in math italic; (2) the `terminals` grammar, overriding the default typesetting for terminals `\` and `-->` by λ and $\longrightarrow$; and (3) `{{ com ...}}` comments, annotating productions and judgements.

One can also write **tex** annotations to override the default typesetting at the level of productions, not just tokens. For example, in $F_{<:}$ one might wish to typeset term abstractions with λ and type abstractions with Λ, and fine-tune the spacing, writing productions

```
| \ x : T . t  :: :: Lam      {{ tex \lambda [[x]] \mathord{:} [[T]]. \, [[t]] }}
| \ X <: T . t :: :: TLam     {{ tex \Lambda [[X]] \mathord{<:} [[T]]. \, [[t]] }}
```

to typeset terms such as `(\X<:T11.\x:X.t12) [T2]` as $(\,\Lambda X{<:}T_{11}.\,\lambda x{:}X.\,t_{12}\,)\,[\,T_2\,]$. These annotations define clauses of functions from symbolic terms to the character string of generated LaTeX, overriding the built-in default clause. Similarly, one can control typesetting of symbolic metavariable and nonterminal roots, e.g. to typeset a nonterminal root `G` as Γ.

Concrete terms To fully specify the concrete syntax of the object language one need only add definitions for the lexical form of variables, concrete instances of metavariables, with the `{{ lex alphanum}}` hom in the **metavar** block. Here **alphanum** is a built-in regular expression. Concrete examples can then be parsed by the tool and pretty-printed into LaTeX or proof assistant code.

List forms For an example that is rather more typical of a large-scale semantics, consider the record typing rule shown in the top half of Fig. 3.3, taken from our OCaml fragment definition. The first, second, and fourth premises are uses of judgement forms; the other premises are uses of `formula` productions with meanings defined by homs. The rule also involves several *list forms*, indicated with dots '...', as is common in informal mathematics. Lists are ubiquitous in programming language syntax, and this informal notation is widely used for good reasons, being concise and clear. We therefore support it directly in the metalanguage, making it precise so that we can generate proof assistant definition clauses, together with the LaTeX shown.

The bottom half of Fig. 3.3 shows the source text for that rule — note the close correspondence to the typeset version, making it easy to read and edit. Looking at it more closely, we see *index variables* n, m, and l occurring in suffixes. There are symbolic nonterminals and metavariables indexed in three different ranges: $e_{\Box}$, $t_{\Box}$, and $\mathit{field\_name}_{\Box}$ are indexed from 1 to n, $\mathit{field\_name}'_{\Box}$ is indexed from 1 to m, and $t'_{\Box}$ is indexed 1 to l. To parse list forms involving dots, the tool finds subterms which can be antiunified by abstracting out components of suffixes.

With direct support for lists, we need also direct support for symbolic terms involving list projection and concatenation, e.g. in the rules below (taken from a different case study).

$$\frac{}{\{\, l'_1 = v_1\,,\, ..\,,\, l'_n = v_n \,\}\,.\, l'_j \;\longrightarrow\; v_j}\quad \textsc{Proj}$$

$$\frac{t \;\longrightarrow\; t'}{\begin{array}{l}\{\, l_1 = v_1\,,\, ..\,,\, l_m = v_m\,,\, l = t\,,\, l'_1 = t'_1\,,\, ..\,,\, l'_n = t'_n \,\} \\ \longrightarrow\; \{\, l_1 = v_1\,,\, ..\,,\, l_m = v_m\,,\, l = t'\,,\, l'_1 = t'_1\,,\, ..\,,\, l'_n = t'_n \,\}\end{array}}\quad \textsc{Rec}$$

Lastly, one sometimes wants to write list *comprehensions* rather than dots, for compactness or as a matter of general style. We support comprehensions of several forms, e.g. with explicit

index i and bounds 0 to $n-1$, as below, and with unspecified or upper-only bounds.

$$\frac{\Gamma \vdash t : \{ \overline{l_i : T_i}^{\,i \in 0..n-1} \}}{\Gamma \vdash t \,.\, l_j : T_j} \quad \textsc{Proj}$$

Other types commonly used in semantics, e.g. finite maps or sets, can often be described with this list syntax in conjunction with type and metaproduction homs to specify the proof assistant representation.

3.2 Binding specifications and substitutions

How to deal with *binding*, and the accompanying notions of substitution and free variables, is a key question in formalised programming language semantics. It involves two issues: one needs to fix on a class of binding structures being dealt with, and one needs proof-assistant representations for them.

The latter has been the subject of considerable attention, with representation techniques based on names, De Bruijn indices, higher-order abstract syntax (HOAS), locally nameless terms, nominal sets, and so forth, in various proof assistants. The annotated bibliography by Charguéraud [31] collects around 40 papers on this, and it was a central focus of the POPLmark challenge [15].

Almost all of this work, however, deals only with the simplest class of binding structures, the *single binders* we saw in the lambda abstraction production of the Fig. 3.1,

$$\mathit{term}, t \ ::= \ \ldots \mid \lambda\, x \,.\, t \quad \mathsf{bind}\ x\ \mathsf{in}\ t$$

in which a single variable binds in a single subterm. Realistic programming languages often have much more complex binding structures, e.g. structured patterns, multiple mutually recursive `let` definitions, comprehensions, or-patterns, and dependent record patterns. We therefore turn our attention to the potential range of binding structures.

We introduce a novel metalanguage for specifying binding structures, expressive enough to cover all the above but remaining simple and intuitive.

The binding metalanguage comprises two forms of annotation on productions. The first, $\mathsf{bind}\ \mathit{mse}\ \mathsf{in}\ \mathit{nonterm}$, is used in the lambda production above. That production has a metavariable x and a nonterminal t, and the binding annotation expresses that, in any concrete term of this production, the variable in the x position binds in the subterm in the t position. A variable can bind in multiple subterms, as in the example of a simple recursive **let** below.

$$\begin{array}{ll} t ::= & \\ \quad \mid \ \mathbf{let\,rec}\, x = t\, \mathbf{in}\, t' & \mathsf{bind}\ x\ \mathsf{in}\ t \\ & \mathsf{bind}\ x\ \mathsf{in}\ t' \end{array}$$

In general a production may require more than just a single variable to bind, and so in the general case *mse* ranges over *metavariable set expressions*, which can include the empty set, singleton metavariables (e.g. the x above, implicitly coerced to a singleton set), and unions.

More complex examples require one to collect together sets of variables. For example, the grammar below has structured patterns, with a $\mathbf{let}\, p = t\ \mathbf{in}\ t'$ production in which all the binders of the pattern p bind in the continuation t'.

t ::=		
	x	
	(t_1, t_2)	
	$\mathbf{let}\, p = t\, \mathbf{in}\, t'$	bind binders(p) in t'
p ::=		
	_	binders = {}
	x	binders = x
	(p_1, p_2)	binders = binders(p_1) $\cup$ binders(p_2)

Here the bind clause binds all of the variables collected as binders(p). We see a user-defined *auxiliary function* called binders, which is defined by structural induction over patterns p to build the set of variables mentioned in a pattern. The clauses that define the binders auxiliary are the second form of binding annotation. For example binders(x) is the singleton set {x}, while binders(((x, x), y)) is the set {x, y}. A definition may involve many different auxiliary functions; "binders" is a user identifier, not a keyword. The tool supports binding for the list forms: metavariable set expressions can include lists of metavariables and auxiliary functions applied to lists of nonterminals, e.g. as in the record patterns below.

p ::=		
	x	b = x
	$\{l_1 = p_1, .., l_n = p_n\}$	b = b(p_1..p_n)

This suffices to express the binding structure of almost all the natural examples we have come across, including definitions of mutually recursive functions with multiple clauses for each, join-calculus definitions [39], dependent record patterns, and many others.

Given a binding specification, the tool can generate substitution functions automatically. Fig. 3.1 contained the block:

```
substitutions
  single term var :: tsubst
```

which causes Ott to generate proof-assistant functions for single substitution of term variables by terms over all (non-subgrammar) types of the grammar — here that is just `term`, and a substitution function named `tsubst_term` is generated. Multiple substitutions can also be generated, and there is similar machinery for free variable functions.

The syntax of a precise fragment of the binding metalanguage is given in [75], where we have used Ott to define part of the Ott metalanguage. A simple type system enforces sanity properties, e.g. that each auxiliary function is only applied to nonterminals that it is defined over, and that metavariable set expressions are well-sorted, not mixing distinct classes of variables. Interestingly, the Ott binding specification language has been integrated in the latest Nominal Isabelle implementation [103].

3.3 Syntactic Design

Some interlinked design choices keep the metalanguage general but syntactically lightweight. Issues of concrete syntax are often best avoided in semantic research, tending to lead to heated and unproductive debate. In designing a usable metalanguage, however, providing a lightweight syntax is important, just as it is in designing a usable programming language. We aim to let the working semanticist focus on the content of their definitions without being blinded by markup, inferring data that can reasonable be inferred while retaining enough redundancy that the tool can do useful error checking of the definitions. Further, the community has developed a variety of well-chosen concise notations; we support some (though not all) of these. The tradeoffs are rather different from those for conventional programming language syntax.

There are no built-in assumptions on the structure of the mathematical definitions (e.g., we do not assume that object languages have a syntactic category of expressions, or a small-step reduction relation). Instead, the tool supports definitions of arbitrary syntax and of inductive relations over it. Syntax definitions include the full syntax of the symbolic terms used in rules (e.g. with metaproductions for whatever syntax is desired for substitution). Judgements can likewise have arbitrary syntax, as can formulae. To our surprise, meta-productions coupled with arbitrary homomorphisms have been used as hooks to overcome limitations of the default theorem proving encodings, and revealed an extremely flexible mechanism.

The tool accepts arbitrary context-free grammars, so the user need not go through the contortions required to make a non-ambiguous grammar (e.g. for `yacc`). Abstract syntax grammars, considered concretely, are often ambiguous, but the symbolic terms used in rules are generally rather small, so this ambiguity rarely arises in practice. Where it does, we let the user resolve it with production-name annotations in terms. The tool finds all parses of symbolic terms, flagging errors where there are multiple possibilities. It uses a GLR parser extended with support for list forms (and initially it relied on a scannerless memoized CPS'd parser combinators, taking ideas from [51]), which is simple and sufficiently efficient.

Naming conventions for symbolic nonterminals and metavariables are rigidly enforced — they must be composed of one of their roots and a suffix. This makes many minor errors detectable, makes it possible to lex the suffixes, and makes parsing much less ambiguous.

3.4 Ott in the workflow of a PL researcher

We wrote this paper despite using Ott.

Simon Peyton-Jones, 2010

Despite the above quote, several researchers have incorporated Ott in their workflow. I report below some excerpts of a talk by Stephanie Weirich (U. Penn) [106], because these shed a novel light on the relevance and importance of tool support for semantics.

> *I plan to use Ott in every new paper that I write, in some form. The tool has become an important part of my design process, and I have come to rely on it.*

The purpose of this (part of) the talk is to explain why. [..] This talk is not about the mechanical formalisation of programming language metatheory. *Ott provides a range of uses and, although my coauthors and I have used Coq to prove properties about language specifications generated by Ott, this is not my main mode of use. Instead, the majority of the benefit that I get from Ott is the mechanical formalisation of programming language* specifications.

By specifying the semantics of a programming language (or a simple toy calculi) in an Ott file, then language design becomes a tool-assisted activity instead of pure mathematics. The Ott file can be part of a version repository, so several (geographically distributed) coauthors can work on the design simultaneously, using the most up-to-date definitions. The LaTeX output means that not all coauthors need to understand the Ott input language. Rules are organised and consistently named, so the language specification is concentrated in the Ott files, not scattered and duplicated across a number of tex files.

The process of specifying a language using Ott provides a lightweight form of consistency checking. Definitions in the semantics must parse, ruling out typos and unintentional ambiguity. Notations and metaproductions give flexibility to the specification, while still leaving traces in the Ott input so they cannot be completely informal. Further consistency checking comes from proof assistant code generation—then not only must the definitions parse, they also must typecheck. These consistency checks aid collaboration as much as the final presentation of the material for publication.

The primary advantage that Ott gives is flexibility in the design process. With this flexibility, I can search a much larger space of potential designs more effectively. Part of this flexibility is due to flexible grammars: syntactic changes are often one line changes to the Ott file. (And, I hate to admit it, but changing the syntax of an object language can often lead new insight into its design.)

However, part of the flexibility is due to the consistency checks. Just as typed languages (such as ML and Haskell) are easier to refactor because the type checker helps to identify all of the places in the source code that changes are needed, Ott can identify all of the ramifications of specificational changes. This makes it difficult to miss unintended consequences of such changes. As the system evolves, I do not reprove all of the properties that I think it should have, but I do appreciate the opportunity to reexamine all of the parts of the specification that might invalidate those properties.

Certainly, this process does not provide as much confidence in the correctness of the design as mechanical proofs of metatheory, but it requires much less effort and can be extended to a mechanical proof at a later date. Although the LaTeX output may not be as beautiful (or concise) as in a hand-crafted paper, the real benefits for collaboration and exploration are worth the trouble, and in the end, lead to better designs.

3.5 The way forward

There are many challenges for the future: areas where existing tools (including Ott) are lacking.

Parsing and Pretty-Printing Ott takes a user specification of an arbitrary context-free grammar (with subgrammars and list forms) and builds a parser, to use for parsing semantic rules and examples. However, Ott does not build a standalone and production-quality parser that could be used in a full-scale language implementation; nor does it build a standalone pretty-printer for abstract syntax terms.

Semantics without Syntax Ott shines in cases where the semantics of the object language is expressed principally in terms of a free syntax, e.g. for structured operational semantics and type systems. Outside that domain, e.g. when one deals with the sequential semantics of machine code (with little syntax but much bit manipulation) or with axiomatic relaxed-memory concurrency semantics (expressed with first-order axioms about relations over events), it gives little or no benefit. Instead, one needs good libraries for finite sets, lists, and so on.

The Ott Type System Considered as a type system, Ott grammars can make use of mutually recursive labelled sums-of-products, with subtyping arising from subgrammar declarations (e.g. for a *value* subgrammar of some *expressions*). This serves surprisingly well, but when one wants to start defining functions one quickly also wants top-level parametric polymorphism and perhaps also type classes.

Binding One of the starting points for the Ott development (which began in late 2004), was the realisation that dealing with rich forms of binding becomes important when one goes beyond small calculi; it introduced a broad class of binding specifications. Implementing that (up to alpha conversion) in full generality remains a challenge, and is perhaps too much to aim for — but Ott can now generate the Locally Nameless representation in relatively simple cases (with further proof infrastructure provided by Aydemir and Weirich's LNgen tool [62]). The Nominal Isabelle system now has direct support for a moderately large subset of Ott-like binding specifications.

However, while dealing with binding is certainly essential for some applications, we find many in which it is not an important issue. For example, in our $\text{OCaml}_{\text{light}}$ semantics we could use the fully concrete representation except for a very modest De Bruijn encoding for type variable binders, and in current work on processor semantics there is no binding whatsoever.

Executable Semantics The last part of the POPLmark challenge focussed on making a semantics executable in some form. We would like to re-emphasise its importance: in our view, two primary uses of a semantic definition should be (a) exploring its consequences on examples, at design-time, and (b) testing conformance between it and an implementation (until the day when full compiler verification becomes routine).

Ongoing work on the Lem tool [56, 71] addresses some of these issues. Semantically, we have designed Lem to be roughly the intersection of common functional programming languages and higher-order logics, as we regard this as a sweet spot: expressive enough for the applications we mention above, yet familiar and relatively easy to translate into the various provers; there is intentionally no logical novelty here. Lem has a simple type theory with primitive support for recursive and higher-order functions, inductive relations, n-ary tuples, algebraic datatypes, record types, type inference, and top-level polymorphism. It also includes a type class mechanism broadly similar to Isabelle's and Haskell's (without constructor classes). It differs from the internal logics of HOL4 or Isabelle/HOL principally in having type, function and relation definitions as part of the language rather than encoded into it. Syntactically, Lem resembles OCaml, giving us a popular and readable syntax. For example, here is an extract from a model of the IBM Power multiprocessor architecture [80] that was developed using Lem.

```
let write_reaching_coherence_point_action m s w =
  let writes_past_coherence_point' =
    s.writes_past_coherence_point union {w} in
  let coherence' = s.coherence union
    { (w,wother) | forall (wother IN (writes_not_past_coherence s)) |
      (not (wother = w)) && (wother.w_addr = w.w_addr) } in
  <| s with coherence = coherence';
            writes_past_coherence_point = writes_past_coherence_point' |>

let sem_of_instruction i ist =
  match i with
  | Padd set rD rA rB -> op3regs Add set rD rA rB ist
  | Pandi rD rA simm -> op2regi And SetCR0 rD rA (intToV simm) ist
  endx
```

From this, Lem generates OCaml, HOL4 and Isabelle code, while a Coq backend is in development.

Lem shares many of the goals of our Ott tool: both emphasise source readability, and multi-prover compatibility. However, Lem is a general-purpose specification language, whereas Ott is a domain-specific language for writing specifications of programming languages (i.e., inductive relations over syntax). Thus, Ott supports rich user-defined syntaxes, whereas Lem supports functional programming idioms. Lem and Ott are complementary; we eventually hope to merge the two projects by having Ott generate Lem specifications, instead of Coq, HOL4, and Isabelle itself.

Chapter 4

Integrating typed and untyped code in a scripting language

In which we heretically argue that static typing might not be the ultimate programming language design, and in which we want to have the cake and eat it too.

Problem Scripting languages facilitate the rapid development of fully functional prototypes thanks to powerful features that are often inherently hard to type. A lax view of what constitutes a valid program allows execution of incomplete programs, a requirement of test-driven development. The absence of types also obviates the need for early commitment to particular data structures and supports rapid evolution of systems. However, as programs stabilise and mature—e.g. a temporary data migration script finds itself juggling with the pension benefits of a small country [99]—the once liberating lack of types becomes a problem. Untyped code, or more precisely dynamically typed code, is hard to navigate, especially for maintenance programmers not involved in the original implementation. The effects of refactoring, bug fixes and enhancements are hard to trace. Moreover performance is often not on par with more static languages. A common way of dealing with this situation is to rewrite the untyped program in a statically typed language such as C# or C++, but this is usually costly and far from guaranteed to succeed [107].

The dream We would love to gradually evolve a prototype into a fully-fledged program within the same language. The typed parts of the code should benefit of the usual features of strongly-typed code (static error checking, optimised compilation), the untyped parts should enjoy the flexibility of dynamically typed code, while passing values across the type boundaries should be seamless.

4.1 The design space

Unsurprisingly this dream has been a long standing challenge in the dynamic language community [100, 12, 95, 101, 49, 24]. We start by recalling closely related work through a series of examples. We use as a vehicle for our experiments an object-oriented scripting language called Thorn [21], which runs on a JVM and ought to support the integration of statically and dynamically typed code. The statically typed part of Thorn sports a conventional nominal

type system with multiple subtyping akin to that of Java. Thorn has also a fully dynamic part, where every object is of type `dyn` and all operations performed on `dyn` objects are checked at run-time.

The Typing of a Point. In a language that supports rapid prototyping, it is sometimes convenient to start development without committing to a particular representation for data. Declaring a two-dimensional `Point` class with two mutable fields `x` and `y` and three methods (`getX`, `getY`, and `move`) can be done with every variable and method declaration having the (implicit) type `dyn`. Run-time checks are then emitted to ensure that methods are present before attempting to invoke them.

```
class Point(var x, var y) {
  def getX() = x;
  def getY() = y;
  def move(p) { x:=p.getX(); y:=p.getY() }
}
```

As a first step toward assurance, the programmer may choose to annotate the coordinates with concrete types, say `Int` for integer, but leave the `move` method unchanged allowing it to accept any object that understands `getX()` and `getY()`. The benefit of such a refactoring is that a compiler could emit efficient code for operations on the integer fields. As the argument to `move` is untyped, casts may be needed to ensure that values returned by the getter methods are of the right type.

```
class Point(var x: Int, var y: Int) {
  def getX(): Int = x;
  def getY(): Int = y;
  def move(p){ x:= (Int)p.getX(); y:= (Int)p.getY()}
}
```

Of course, this modification is disruptive to clients of the class: all places where `Point` is constructed must be changed to ensure that arguments have the proper static type. In the long run, the programmer may want more assurance for invocations of `move()`, e.g., by annotating the argument of the method as `pt:Point`. This has the benefit that the casts in the method's body become superfluous. This has the drawback that all client code must (again) be revisited to add static type annotations on arguments and decreases flexibility of the code, as clients may call `move` passing an `Origin` object.

```
class Origin {
   def getX(): Int = 0;
   def getY(): Int = 0;
}
```

While not a subclass of point, and thus failing to type check, `Origin` has the interface required by the method. This is not unusual in dynamically typed programs. Part of the last issue could be somewhat mitigated by the adoption of structural subtyping [27]. This would lift the requirement that argument of `move` be a declared subtype `Point` and would accept any type with the same signature. Unfortunately, this is not enough here, as `Origin`

is not a structural subtype either. The solution to this particular example is to invent a more general type, such as `getXgetY` which has exactly the interface required by `move`.

```
class getXgetY {
  def getX(): Int;
  def getY(): Int;
}
```

This solution does not generalise as, if it was applied systematically, it would give rise to many special purpose types with little meaning to the programmer. A combination of structural and intersection types are often the reasonable choice when starting with an existing untyped language such as Ruby, JavaScript or Scheme (see for example [41, 101]) but they add programmer burden, as a programmer must explicitly provide type declarations, and are brittle in the presence of small changes to the code. For these reasons, Typed Scheme is moving from structural to nominal typing.[1]

Soft Typing An early attempt at bridging the gap between dynamic and static typing is the *soft typing* proposed by Cartwright and Fagan [29] (but can be traced to early work by Cartwright [28]) and subsequently applied to a variety of languages [41, 70, 55, 11, 26, 108, 55, 11]. Soft typing tries to transparently superimpose a type system on unannotated programs, inferring types for variables and functions. On our example, a soft-typing system would infer a type such as `getXgetY` above without programmer intervention, obviating the need to litter the code with overly specific types. When an operation cannot be typed, a dynamic check is emitted and, possibly, a warning for the programmer. A compiler equipped with a soft type checker would never reject a program, preserving expressiveness of the dynamically typed language. The main benefit of soft typing is the promise of warnings for potentially dangerous constructs and the elimination of run-time checks when the compiler can show that an operation is safe. Its drawback is the lack of static guarantee that a given piece of code is free of errors. It is thus not possible for programmers to take key pieces of their system and "make" them safe, or fast. A spelling mistake in a method name will generate a constraint that cannot be satisfied but will only be caught when the method is invoked by client code and in general inferred types can easily get unwieldy and hard to understand for a human programmer. The performance model is opaque as a small change in the code can have a large impact on performance simply because it prevents the compiler from optimising an operation in a hotspot.

Gradual typing Incremental typing schemes have been explored by Bracha and Griswold in Strongtalk [24] which inspired pluggable types [23], in various gradual type systems [12, 93, 97, 96, 45], and recently Typed Scheme [101, 100].

The gradual typing approach of Siek and Taha allows for typed and untyped values to commingle freely [93]. When an untyped value is coerced, or cast, to a typed value, a *wrapper* is inserted to verify that all further interactions through that particular reference behave according to the target type's contract. At the simplest a wrapper is a cast $\langle T \Leftarrow R \rangle$ saying, intuitively, that the value was of type R and must behave as a value of type T.

[1] Matthias Felleisen, presentation at the STOP'09 (Script to Program Evolution) workshop.

The number of wrappers is variable and can, in pathological cases, be substantial [45]. In practice, any program that has more than a single wrapper for any value is likely to be visibly slower. In the presence of aliasing and side-effects the wrappers typically can not be discharged on the spot and have to be kept as long as the value is live. The impact of this design choice is that any operation on a value may fail if that value is a dynamic type which does not abide by the contract imposed by its wrapper. Wrapper have to be manipulated at run-time and compiler optimizations are inhibited as the compiler has to emit code that assumes the presence of wrappers everywhere. Some of these problems may be avoided with program analysis, but there is currently no published work that demonstrates this.

To provide improved debugging support researchers have investigated the notion of *blame control* in the context of gradual typing, [36, 100, 105, 96]. The underlying notion is that concretely typed parts of a program should not be blamed for run-time type errors. As an example, let `T` be a type with a method `m` and `x` be a variable of type `T`. Now, if some object `o`, that does not understand `m`, is stored in `T`, blame tracking will not blame the call `x.m()`—which is correct as `x` has type `T`—for throwing a "message not understood" exception at run-time. Rather, it will identify the place in the code where `o` was cast to `T`. Fine-grained blame control requires that a reference "remembers" each cast it flows through, perhaps modulo optimizations on redundant casts. Storing such information in references and not in objects is key to achieve traceability, but incurs additional run-time overhead on top of the run-time type checks. Evaluating the performance impact of blame tracking and its practical impact on the ability to debug gradually typed programs has not yet been investigated. We use the term gradual typing to refer to a family of approaches that includes hybrid typing [37] and that have their roots in a contract-based approach of [36, 42].

Our design choices Most of the previous work which had its root in dynamically typed languages (Smalltalk, Scheme, Ruby and JavaScript) and tried to provide static checking. On the contrary, we would like to provide the flexibility of dynamic languages to static languages. At the language level, we are thus willing to forgo some of the most dynamic features of languages, such are run-time modification of object interfaces, in languages like JavaScript or Ruby. At the implementation level, the addition of opcodes to support dynamic languages in Java virtual machines makes it possible to envision mixing typed and untyped code without sacrificing performance. The research question is thus how to integrate these different styles of programming within the same language. In particular, it would not be acceptable for statically typed code to either experience run-time failures or be compiled in a less efficient to support dynamic values. Conversely, the expressiveness of dynamic parts of the system should not be restricted by the mere presence of static types in unrelated parts of the system.

4.2 A Type System for Program Evolution

We propose a type system for a class-based object-oriented programming language with three kinds of types. *Dynamic types*, denoted by the type `dyn`, represent values that are manipulated with no static checks. Dynamic types offer programmers maximal flexibility as any operation is allowed, as long as the target object implements the requested method.

However, `dyn` gives little aid to find bugs, to capture design intents, or to prove properties. At the other extreme, we depart from previous work on gradual typing, by offering *concrete types*. Concrete types behave exactly how programmers steeped in statically typed languages would expect. A variable of concrete type `C` is guaranteed to refer to an instance of class `C` or one of its subtypes. Concrete types drastically restrict the values that can be bound to a variable as they do not support the notion of wrapped values found in gradual type systems. Concrete types are intended to facilitate optimizations such as unboxing and inlining as the compiler can rely on the static type information to emit efficient code. Finally, as an intermediate step between the two, we propose *like types*. Like types combine static and dynamic checking in a novel way. For any concrete type `C`, there is a corresponding like type, written `like C`, with an identical interface. Whenever a programmer uses a variable typed `like C`, all manipulations of that *variable* are checked statically against `C`'s interface, while, at run-time, all uses of the value bound to the variable are checked dynamically. Figure 4.1 shows the relations between types (`dyn` will be implicit in the code snippets). Full arrows indicate traditional subtype relations (so, for instance if `B` is a subtype of `A`, then `like B` is a subtype of `like A`), dotted lines indicate implicit `dyn` casts, and finally, dashed lines show situations where `like` casts are needed. Observe that the classes `C` and `D` are unrelated by inheritance.

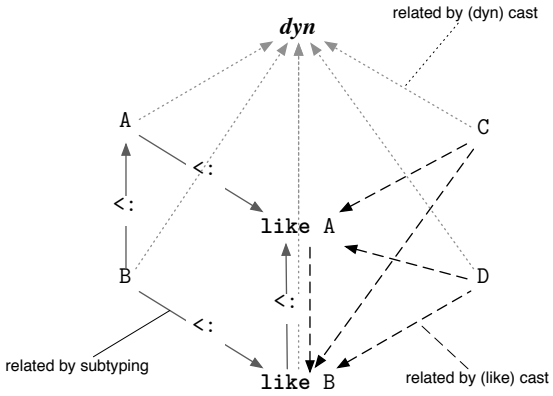


Figure 4.1: Type Relations.

In our design we have chosen a nominal type system, thus subtype relation between concrete types must be explicitly declared by `extends` clauses. While we believe that our approach applies equally well to structural types, our choice is motivated by pragmatic reasons. Using class declarations to generate eponymous types is a compact and familiar (to most programmers) way to construct a type hierarchy. Moreover, techniques for generating efficient field access and method dispatch code sequences for nominal languages are well known and supported by most virtual machines.

The first key property of like type annotations is that they are local. This is both a strength and a limitation. It is a strength because it enables purely local type checking. Returning to our example, like types allow us to type the parameter to `move` thus:

```
def move(p: like Point) {
  x := p.getX(); y := p.getY();
  p.hog();     # !Raises a compile time error!
}
```

Declaring the variable `p` to be `like` a `Point`, makes the compiler check all operations on that variable against the interface of `Point`. Thus, the call to `hog` would be statically rejected since there is no such method in `Point`. The annotation provides the static information

necessary to enable IDE support commonly found in statically typed languages (but not in dynamic ones).

The second key property is that like types do not restrict flexibility of the code. Declaring a variable to be `like C` is a promise on how that *variable* is used and not to what *value* that variable can be bound to. For the client code, a like typed parameter is similar to a `dyn`. The question of when to test conformance between a variable's type and the value it refers to is subtle. One of our goals was to ensure that the addition of like type annotations would not break working code. In particular, adding type annotations to a library class should not cause all of its clients to break. So instead of checking at invocation time, each use of a like typed variable is preceded by a check that the target object has the requested method. If the check fails, a run-time exception is thrown. Consider the `Coordinate` class, which is similar to `Point`, but lacks a `move` method:

```
class Coordinate(var x: Int, var y: Int) {
  def getX(): Int = x;
  def getY(): Int = y;
}
```

In our running example, if `move` expects a `like Point`, then calling `move` with a `Coordinate` works exactly as in an untyped language. Even if `Coordinate` does not implement the entire `Point` protocol, it implements the *relevant* parts, the methods needed for `move` to run successfully. If it lacked a `getY` method, passing a `Coordinate` to `move` would compile fine, but result in an exception at run-time. More interestingly, `move` can also accept an untyped definition of `Coordinate`:

```
class Coord(x,y) {  def getX() = x;  def getY() = y;  }
```

Here, the run-time return value of `getX` and `getY` are tested against `Int`: invoking `move` with the argument `Coord(1,2)` would succeed, `Coord("a","b")` would raise an exception. Observe that if `Point` used `like Int`, checking the return type would not be necessary as assigning to a like type always succeeds.

Interfacing typed and untyped code. Consider a call `p1.move(p2)` with different declared types for variables `p1`, `p2` and `pt` (the type of the parameter in the `move` method). Depending on the static type information available on the receiver, different static checks are enabled, and different run-time checks are needed to preserve type-safety. We go through these in detail in Figure 4.2.

Assume that the parameter `pt` in `move` has type `dyn`, then all configurations of receiver and argument are allowed and will compile successfully. In case the parameter has the type `like Point`, again, all configurations are statically valid. The last case to consider is when `pt` has the concrete type `Point`. In that case, there are several subcases that need to be looked at. If the receiver `p1` is untyped, then, as expected, no static checks are possible. At run-time, we must consequently check that `p1` understands the `move` method and if so, that `p2`'s run-time type satisfies the type on the parameter in the `move` method. Since, `pt` is `Point`, a subtype test will be performed at run-time. If the receiver `p1` is a concrete type, the type of the argument `p2` will be statically checked: if it is `dyn`, a compile-time error will be reported; if it is `like Point`, the compiler will accept the call and emit a run-time subtype

p1	p2	pt	Result
_	_	dyn	OK
_	_	like Point	OK
dyn	_	Point	OK
Point	dyn	Point	ERR
Point	like Point	Point	OK *
Point	Point	Point	OK
like Point	dyn	Point	ERR
like Point	like Point	Point	OK *
like Point	Point	Point	OK

Figure 4.2: Configurations of declared types. The column labeled Result indicate if there will be a compile-time error. The formalisation reported in [109] is slightly more strict and requires explicit casts in cases labeled *.

test; if `p2` is a `Point` a straightforward typed invocation sequence can be emitted. Finally, the case where the receiver is declared `like Point` is similar to the previous case, with the exception that a run-time test is emitted to check for the presence of a `move` method in `p1`.

If `move` had some concrete return type `C`, invoking it on a like typed receiver, would then check that the value returned from the method was indeed a (subtype of) `C`. If this cannot be determined statically, for instance if the actual method does not return a concrete type, then a type test is performed on the value returned. Calls with untyped receivers never need to type-check return values, as client code has no expectations that must be met. The concretely typed case follows from regular static checking.

Revisiting a previous example, consider a variant of `move` with a call to `getY` guarded by an `if` and assume that `p` is bound at run-time to an object that does not have a `getY`.

```
def move(p: like Point) {
  x := p.getX();
  if (unlikely) y := p.getY();
}
```

As the system only checks uses of `p`, the error triggers if the condition is true. Some situations, which are hard to type in systems that perform eager subtype tests, e.g., at the start of the method call, work smoothly thanks to this lazy checking. As a result like types are not structural, but "semi-structural" since they only require the methods called to be present.

Code evolution. Like types provide an intermediate step between dynamic and concrete types. In some cases the programmer might want to replace `like C` annotations with concrete `C` annotations, but this is not always straightforward. The reason is the shift in notion of subtype—from (a variant on) structural to nominal. Fortunately, studies of the use of dynamic features in practice in dynamically typed programs [14, 48] suggest that many dynamic programs are really not that polymorphic. When this is the case, the transition is as simple as removing the `like` keyword. Changing a piece of code that is largely like typed

to use concrete types imposes an additional level of strictness on the code. Subsequently, stores from like typed (or `dyn`) variables into concretely typed variables must be guarded by type checks. The Thorn compiler inserts these checks automatically where needed and prints a warning to avoid suppressing actual compile-time errors. Notably, when accessing a concretely typed field or calling a method with concrete return type on a like typed receiver, the resulting value will be concretely typed. Subsequent operations on the returned value will enjoy the same strict type checking as all concrete values and can be compiled more efficiently than operations on like typed receivers.

In some cases, one can imagine going from typed code to untyped, for example to facilitate interaction with some larger untyped program, or to increase the flexibility in the code. Simply adding a `like` keyword in the relevant places, e.g., in front of types in the interface, or on key variables, immediately allows for a higher degree of flexibility without losing the local checking and still keeping the design intent in the code.

Compile-Time Optimizations In Thorn, all method calls go through a dispatching function. With like types, three different dispatching functions are used to perform the necessary run-time checks described above. Every user written method call is compiled down to one of those dispatching functions depending on the type information available at the call-site. The dispatching function used for untyped calls performs run-time type checks and unboxes boxed primitives. The like typed dispatching function checks that the intended method is actually present in the receiver and has compatible types. The concretely typed dispatching function performs a simple and fast lookup (as e.g. in Java), knowing that the method is present. Additionally, if the static type of the argument is a like type when some concrete type is expected, the Thorn compiler will insert a run-time type test and issue a warning.

Like types allow interaction with an untyped object through a typed interface and guarantees that operations that succeed satisfy the typing constraints specified in the interface. Consider the following code snippet that declares two cells—one for untyped content and one for integers:

```
class Cell(var x) {
  def get() = x;
  def set(x') { x:= x' }
}
class IntCell(var i: Int) {
  def get(): Int = i;
  def set(j:Int) { i:= j }
}
box = Cell(32);
y = box.get();
ibox: like IntCell = box;
z: Int = ibox.get();
ibox.set(z+10);
```

If `ibox.get()` succeeds, we statically know its return type to be an `Int` since the cell is accessed through a like typed interface. Subsequent operations on `z` enjoy static type checking and can be optimized, contrarily to uses of `y`. For example, the `+` operation on the

last line can be compiled into machine instructions or equivalent, rather than a high-level method call on an integer object. Alternatively, the programmer might explicitly cast `y` to `Int`. However, typing the cell `like IntBox` type checks all interactions with the cell statically and gives static type information about what is put into and taken from it: this requires a single annotation at a declaration rather than casts spread all over the code.

Relating Like Types to Previous Work Like types add local checking to code without restricting its use from untyped code. In contrast to gradual typing [12, 93, 97, 96, 45] and pluggable types [23], it introduces an intermediate step on the untyped–concretely typed spectrum and uses nominal rather than structural subtyping. Furthermore, it only requires operations to be present when actually used. As a result, operations on concrete types can be efficiently implemented and like types used where flexibility is desired. Typed Scheme [100, 101] uses contracts on a module level, rather that simple type annotations, and does not work with object structures. Soft typing [29] infers constraints from code, rather than lets programmers expressly encode design intent in the form of type annotations. Adding soft typing to Java [55, 11] faces similar although fewer problems. An important difference between like types and gradual typing systems like $\mathbf{Ob}^{?}_{<:}$ [93], is that code completely annotated with like types can go wrong due to a run-time type error. On the other hand, a code completely annotated with concrete types will not go wrong.

A perhaps unusual design decision is the lack of blame control. If a method fails, e.g., due to a missing method in an argument object, we cannot point to the place in the program that subsequently lead to this problem. In this respect, the blame tracking support offered by like types is not much better than what is offered by a run-time typecast error. This is a design decision. Nothing prevents adding blame control to like types in accordance with previous work (e.g., [94, 8]). The rationale for our design is to avoid performance penalties. Keeping like types blame-free allows for a wrapper-less implementation.

As part of the aborted ECMAScript 4 standard, Cormac Flanagan proposed a type system closely related to the one we present in this paper [38]. The Objective-C language has like types for objects and no concrete object types. Classes can be either `dyn` (called `id`) or like typed, and the compiler warns rather than rejects programs due to other language features that can make non-local changes to classes.

4.3 The way forward

Our design rewards a programmer that makes the effort of writing type annotation with both a stronger security guarantee and faster execution, this differentiates it from the state of the art discussed earlier. However a language design is not complete unless evaluated writing “real” programs on a “real” implementation. If the latter can be provided by the language designers (with lots of efforts), the former requires the language to be adopted by a community. We provided a complete and efficient implementation of our design in the language Thorn; this allowed us to experiment with actual optimisations enabled by the type informations (the simplest being unboxing base values). Unfortunately the Thorn project was abandoned (the industrial partner stepped out, mostly for political reasons) before the

public release of the compiler. As such our design as not been tested on anything bigger than the couple of thousands lines of the wiki implementation described in the paper.

Today JavaScript is de facto the assembler of the web and benefits from state of the art run-time implementations. JavaScript is an highly dynamic language, see [77] for an accurate analysis of its run-time behaviour, but typed extensions, motivated from the need to develop large-scale JavaScript applications appeared recently. For instance the TypeScript language [65] adds optional types, classes and modules to JavaScript, performs error checking, and then compiles away these construct generating plain JavaScript code. A proposal to support for class-based programming has even submitted to the ECMAScript standard. All this suggests that JavaScript is a natural framework in which implement and evaluate the like-type proposal; with Vitek and Richard we are working on the semantic design of a like-JavaScript system and its implementation on top of a commercial JavaScript engine built by Oracle.

More in general, languages like C, C++, Perl and JavaScript (and on a different domain the language R), are here to stay. For a long time these have been criticised and mostly ignored by the programming language research community. It is true that these languages come with their set of weird (and in some case arguably insane) design choices; rather than dismissing them, I believe these languages can be source of formidable challenges for research, and it is our duty to leverage formal methods to make these languages better languages.

Epilogue

The amazing complexity of today's programming calls for new engineering methods to build robust systems. Recent progress in formal methods and mechanised proof assistants have made it possible to apply mathematically rigorous methods to the specification, testing and verification of ambitious projects such as compilers or micro-kernels. Nevertheless, despite some remarkable successes, working with full-scale, realistic, system interfaces is still in its infancy and novel tools and reasoning methods are needed to support a major change in the engineering practice. In this spirit, each chapter of this memoire attempts to give a solution to a *problem* arising from *programming practice*. I believe it is important to avoid oversimplifications, so that the research results, even if not always pretty and elegant, have a chance of being backported to the programming practice. The open research directions discussed constitute necessary and significant steps towards making modern systems easier to program, analyse and test. In the next years I will continue working towards this ambitious goal, tackling the aforementioned research directions or the new exciting problems that will appear along the way, as my life of researcher is full of good surprises.

Bibliography

[1] Linux kernel traffic, 1999. `http://www.kernel-traffic.org/kernel-traffic/kt19991220_47.txt`.

[2] *AMD64 Architecture Programmer's Manual (3 vols)*. Advanced Micro Devices, Sept. 2007. rev. 3.14.

[3] Intel 64 architecture memory ordering white paper, 2007. Intel Corporation. SKU 318147-001.

[4] *AMD64 Architecture Programmer's Manual (5 vols)*. Advanced Micro Devices, May 2013. rev. 3.20.

[5] *Intel 64 and IA-32 Architectures Software Developer's Manual (5 vols)*. Intel Corporation, Sept. 2013. rev. 48.

[6] *Power ISA*. IBM, May 2013. Version 2.07.

[7] A. Adir, H. Attiya, and G. Shurek. Information-flow models for shared memory with an application to the PowerPC architecture. *IEEE Trans. Parallel Distrib. Syst.*, 14(5):502–515, 2003.

[8] A. Ahmed, R. B. Findler, J. Matthews, and P. Wadler. Blame for all. In *STOP*, 2009.

[9] J. Alglave, A. Fox, S. Ishtiaq, M. Myreen, S. Sarkar, P. Sewell, and F. Zappa Nardelli. The semantics of Power and ARM multiprocessor machine code. In *DAMP*, 2009.

[10] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running tests against hardware. In *TACAS*, 2011.

[11] D. Ancona, G. Lagorio, and E. Zucca. Type inference for polymorphic methods in Java-like languages. In *ICTCS*, 2007.

[12] C. Anderson and S. Drossopoulou. BabyJ: From object based to class based programming via types. *Electronic Notes in Theoretical Computer Science*, 82(7), 2003.

[13] ARM. *ARM Architecture Reference Manual (ARMv7-A and ARMv7-R edition)*. April 2008.

[14] J. Aycock. Aggressive type inference. In *International Python Conference*, 2000.

[15] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark Challenge. In *TPHOLs*, 2005.

[16] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *POPL*, 2012.

[17] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, 2011.

[18] P. Becker. *Standard for Programming Language C++ - ISO/IEC 14882*, 2011.

[19] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *ICFP*, 2009.

[20] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[21] B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strniša, J. Vitek, and T. Wrigstad. Thorn–robust, concurrent, extensible scripting on the JVM. In *OOPSLA*, 2009.

[22] H.-J. Boehm and S. Adve. Foundations of the C++ concurrency memory model. In *PLDI*, 2008.

[23] G. Bracha. Pluggable type systems. *OOPSLA04, Workshop on Revival of Dynamic Languages*, 2004.

[24] G. Bracha and D. Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *OOPSLA*, 1993.

[25] S. D. Brookes. Full abstraction for a shared-variable parallel language. *Inf. Comput.*, 127(2):145–163, 1996.

[26] P. Camphuijsen, J. Hage, and S. Holdermans. Soft typing PHP. Technical report, Utrecht University, 2009.

[27] L. Cardelli. Structural Subtyping and the Notion of Power Type. In *POPL*, 1988.

[28] R. Cartwright. User-defined data types as an aid to verifying LISP programs. In *ICALP*, 1976.

[29] R. Cartwright and M. Fagan. Soft Typing. In *PLDI*, 1991.

[30] G. Castagna, J. Vitek, and F. Zappa Nardelli. The Seal calculus. *Inf. Comput.*, 201(1):1–54, 2005.

[31] A. Charguéraud. Annotated bibliography for formalization of lambda-calculus and type theory. `http://arthur.chargueraud.org/projects/binders/biblio.php`, 2006.

[32] The `cmmtest` tool, 2012. `http://www.di.ens.fr/~zappa/projects/cmmtest`.

[33] The Coq proof assistant, v.8.1, 2008. `http://coq.inria.fr/`.

[34] D. Dice. Java memory model concerns on Intel and AMD systems. `http://blogs.sun.com/dave/entry/java_memory_model_concerns_on`, Jan. 2008.

[35] E. Eide and J. Regehr. Volatiles are miscompiled and what to do about it. In *EMSOFT*, 2008.

[36] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP*, 2002.

[37] C. Flanagan. Hybrid type checking. In *POPL*, 2006.

[38] C. Flanagan. ValleyScript: It's like static typing. Technical report, UC Santa Cruz, 2007.

[39] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *CONCUR*, 1996.

[40] C. Fournet, N. Guts, and F. Zappa Nardelli. A formal implementation of value commitment. In *ESOP*, 2008.

[41] M. Furr, J. hoon An, J. Foster, and M. Hicks. Static type inference for ruby. In *SAC*, 2009.

[42] K. E. Gray, R. B. Findler, and M. Flatt. Fine-grained interoperability through mirrors and contracts. In *OOPSLA*, 2005.

[43] N. Guts. *Auditability for security protocols.* PhD thesis, Université Denis Diderot Paris 7, 2011.

[44] N. Guts, C. Fournet, and F. Zappa Nardelli. Reliable evidence: Auditability via typing. In *Esorics*, 2009.

[45] D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. In *TFP*, 2007.

[46] A. Hobor, A. W. Appel, and F. Zappa Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, 2008.

[47] The HOL 4 system, Kananaskis-4 release, 2007. `http://hol.sourceforge.net/`.

[48] A. Holkner and J. Harland. Evaluating the dynamic behaviour of Python applications. In *ACSC*, 2009.

[49] A. S. Inc. ActionScript 3.0 Language and Components Reference, 2008.

[50] Isabelle 2008, 2008. `http://isabelle.in.tum.de/`.

[51] M. Johnson. Memoization in top-down parsing. *Comput. Linguist.*, 21(3):405–417, 1995.

[52] S. Kahrs. Mistakes and ambiguities in the definition of Standard ML. Technical Report ECS-LFCS-93-257, University of Edinburgh, 1993.

[53] A. Kennedy, N. Benton, J. B. Jensen, and P.-E. Dagand. Coq: the world's best macro assembler? In *PPDP*, 2013.

[54] G. Klein, T. Nipkow, and L. Paulson, editors. *The Archive of Formal Proofs*. 2009. `http://afp.sf.net`.

[55] G. Lagorio and E. Zucca. Just: Safe unknown types in Java-like languages. *Journal of Object Technology*, 6(2), 2007.

[56] Lem, a tool for lightweight executable mathematics, v.0.3, 2011. `http://www.cl.cam.ac.uk/~so294/lem/`.

[57] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL*, 2006.

[58] X. Leroy. Compcert, v. 1.5, 2009.

[59] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[60] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[61] X. Leroy et al. The Objective Caml system release 4.01: Documentation and user's manual, 2013.

[62] Lngen: Tool support for locally nameless representations, 2009. `http://www.cis.upenn.edu/~sweirich/papers/lngen/`.

[63] P. Loewenstein. Personal communication, Nov. 2008.

[64] M. Merro and F. Zappa Nardelli. Behavioral theory for mobile ambients. *J. ACM*, 52(6):961–1023, 2005.

[65] Microsoft. Typescript, 2012.

[66] R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

[67] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[68] R. Morisset, P. Pawan, and F. Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *PLDI*, 2013.

[69] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. Rocksalt: better, faster, stronger sfi for the x86. In *PLDI*, 2012.

[70] S.-O. Nyström. A soft-typing system for Erlang. In *Erlang Workshop*, 2003.

[71] S. Owens, P. Böhm, F. Zappa Nardelli, and P. Sewell. Lem: A lightweight tool for heavyweight semantics. In *ITP*, 2011.

[72] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOLs*, 2009.

[73] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO (extended version). Technical Report UCAM-CL-TR-745, Univ. of Cambridge, 2009.

[74] S. Owens, S. Sarkar, and P. Sewell. The x86-TSO model, 2009. `www.cl.cam.ac.uk/users/pes20/weakmemory/`.

[75] G. Peskine, S. Sarkar, P. Sewell, and F. Zappa Nardelli. Binding and substitution (note), 2007. `http://www.cl.cam.ac.uk/users/pes20/ott/`.

[76] S. Peyton Jones, editor. *Haskell 98 Language and Libraries. The Revised Report.* Cambridge University Press, 2003.

[77] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *PLDI*, 2010.

[78] A. Rossberg. Defects in the revised definition of Standard ML. Technical report, Saarland University, 2001. Updated 2007/01/22.

[79] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams. Synchronising C/C++ and POWER. In *PLDI*, 2012.

[80] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *PLDI*, 2011.

[81] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *POPL*, 2009.

[82] J. Sevcik. *Program Transformations in Weak Memory Models.* PhD thesis, University of Edinburgh, 2008.

[83] J. Sevcik. Safe optimisations for shared-memory concurrent programs. In *PLDI*, 2011.

[84] J. Sevcik, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. In *POPL*, 2011.

[85] J. Sevcik, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. Compcerttso: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22, 2013.

[86] P. Sewell. On implementations and semantics of a concurrent programming language. In *CONCUR*, 1997.

[87] P. Sewell, J. J. Leifer, K. Wansbrough, F. Zappa Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. design rationale and language definition. Technical Report UCAM-CL-TR-605, 2004.

[88] P. Sewell, J. J. Leifer, K. Wansbrough, F. Zappa Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. In *ICFP*, 2005.

[89] P. Sewell, J. J. Leifer, K. Wansbrough, F. Zappa Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. *Journal of Functional Programming*, 17(4–5):547–612, 2007.

[90] P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010.

[91] P. Sewell, F. Zappa Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122, 2010.

[92] P. Sewell, F. Zappa Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: Effective tool support for the working semanticist. In *ICFP*, 2007.

[93] J. Siek and W. Taha. Gradual typing for objects. In *ECOOP*, 2007.

[94] J. Siek and P. Wadler. Threesomes, with and without blame. In *STOP*, 2009.

[95] J. G. Siek. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*, 2006.

[96] J. G. Siek, R. Garcia, and W. Taha. Exploring the design space of higher-order casts. In *ESOP*, 2009.

[97] J. G. Siek and M. Vachharajani. Gradual typing with unification-based inference. In *DLS*, 2008.

[98] M. Sperber, R. k. Dybvig, M. Flatt, A. Van straaten, R. Findler, and J. Matthews. Revised^6 report on the algorithmic language Scheme, 2007. `http://www.r6rs.org/`.

[99] E. Stephenson. Perl Runs Sweden's Pension System. O'Reilly Media, 2001.

[100] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *DLS*, 2006.

[101] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *POPL*, 2008.

[102] Twelf 1.5, 2005. `http://www.cs.cmu.edu/~twelf/`.

[103] C. Urban and C. Kaliszyk. General bindings and alpha-equivalence in nominal isabelle. *Logical Methods in Computer Science*, 8(2), 2012.

[104] V. Vafeiadis and F. Zappa Nardelli. Verifying fence elimination optimisations. In *SAS*, 2011.

[105] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *ESOP*, 2009.

[106] S. Weirich, S. Owens, , P. Sewell, and F. Zappa Nardelli. Ott or nott. In *WMM*, 2010.

[107] U. Wiger. Four-fold increase in productivity and quality. In *Workshop on Formal Design of Safety Critical Embedded Systems*, 2001.

[108] A. Wright and R. Cartwright. A practical soft type system for Scheme. In *Conference on LISP and Functional programming*, pages 250–262, 1994.

[109] T. Wrigstad, F. Zappa Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *POPL*, 2010.

[110] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI*, 2011.