Compiling Functional Types to Relational Specifications for Low Level Imperative Code

Nick Benton, Nicolas Tabareau

Compiling Functional Types to Relational Specifications for Low Level Imperative Code

Nick Benton
Microsoft Research, Cambridge
nick@microsoft.com

Nicolas Tabareau
PPS, Université Denis Diderot, Paris
tabareau@pps.jussieu.fr

Abstract

We describe a semantic type soundness result, formalized in the Coq proof assistant, for a compiler from a simple functional language into an idealized assembly language. Types in the high-level language are interpreted as binary relations, built using both second-order quantification and separation, over stores and values in the low-level machine.

Categories and Subject Descriptors F.3.1 [*Logics and meanings of programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification, Specification techniques; F.3.3 [*Logics and meanings of programs*]: Studies of Program Constructs—Type structure; D.3.4 [*Programming Languages*]: Processors—Compilers; D.2.4 [*Software Engineering*]: Software / Program Verification—Correctness proofs, formal methods

General Terms Languages, theory

Keywords Compiler verification, type soundness, relational parametricity, separation logic, proof assistants

1. Introduction

What kinds of correctness properties do we wish to establish of our compilers? The most obvious answer is that when a high-level source program is compiled to produce a low-level target, the behaviour of the target always agrees with a high-level semantics of the source. But we usually also want to be sure that the target code satisfies certain safety or liveness properties, ensuring that 'bad things' don't happen, or that 'good' ones do. Such properties include memory safety, adherence to information flow policies, termination, and various forms of resource boundedness. For applications in language-based security, the good news is that (fancy) type-like properties of this kind, which at least *seem* less complex to state and check than full functional correctness, are the important ones. On the other hand, for such applications one would really like to certify the code that actually runs, which involves formalizing and verifying type-like properties of machine code. How to do that is the problem we address here.

The approach is essentially that of our earlier work on specifying and verifying memory managers (Benton 2006) and type preservation for a simple imperative language (Benton and Zarfaty 2007), and we will not repeat all the methodological arguments here. Briefly, we want to find low-level specifications that should be satisfied by target code compiled from source phrases of particular high-level types. Ideally, these specifications should be both modular and expressed without reference to concepts that are specific to the particular high-level language. This is important: we are not just interested in proving properties of complete, closed programs. We want (and have) to prove things about the result of linking or replacing bits of compiled code with code from elsewhere, including both the runtime system (which may be handcrafted machine code) or code compiled from other high-level languages. For maximum flexibility and strong guarantees, we would like these specifications to be semantic, i.e. defined extensionally in terms of the observable behaviour of programs rather than in terms of a purely syntactic type system for low-level code. One way of characterizing our goal is that we would like to know just what contract should be satisfied by a C or assembly language function that is intended to behave as an ML value of some (possibly higher order) type.

In previous work, we have proposed that these goals can be achieved by giving a semantics for high-level types as relational specifications over low-level code, and we have shown how this works out in the case of a very simple imperative language. The main contribution of the present paper is to show how such a low-level relational interpretation of types can be extended to a language with higher-order functions. The definitions and results presented here have been formalized in the Coq proof assistant. As we will explain, both our general low-level reasoning framework and its encoding in Coq have been improved relative to our earlier work. The Coq development is available from the authors' home pages.

2. Source and Target Languages

2.1 The Low Level Target

The idealized low level machine code into which we compile is the same as in our previous work. There is a single datatype, the natural numbers, though different instructions treat elements of that type as code pointers, heap addresses, integers, etc.. The store is a total function from naturals to naturals and the code heap is a total function from naturals to instructions (immutable and distinct from the data heap). Computed branches and address arithmetic are perfectly allowable. There is no built-in notion of allocation and no notion of stuckness or 'going wrong': the only observable behaviours are termination and divergence. There are no registers; we simply adopt a programming convention of using the first few memory locations in a register-like fashion.

The Coq specification of our machine involves an inductive type of `instructions`, including halting, direct and indirect loads, stores and jumps, arithmetic and tests. Details can be found in the

TLDI'09, January 24, 2009, Savannah, Georgia, USA.

proof scripts or our earlier paper (Benton and Zarfaty 2007); here we just use an obvious pseudocode in which, for example

```
100: [5] ← [[6]+1]    // in C:  *5 = *(*6 + 1);
```

means that the instruction at code address 100 reads the contents of the memory location following that pointed to by location 6, and stores the result in location 5. The mutable state of our machine is specified by

```
Definition state := nat → nat.
Definition program := nat → instruction.
```

and there is then a one-step transition function

```
Definition sem_instr (ins:instruction) (s:state)
  (pc:nat) : option (state * nat) := ...
```

mapping an instruction, state and program counter to either a new state and program counter, or `None` in the case that the instruction is a `halt`. A configuration is thus a triple of a program, a state and a program counter. All our notions of behaviour arise from the primitive notion of termination:

```
Fixpoint kstepterm (k:nat) (p:program) (s:state)
                   (l:nat) {struct k} : Prop :=
 match k with
   | O ⇒ False
   | (S j) ⇒
      match sem_instr (p l) s l with
        | None ⇒ True
        | Some (s', l') ⇒ kstepterm j p s' l'
      end
 end.

Definition terminates p s l :=
                  ∃ k, kstepterm k p s l.
```

So a configuration terminates if there is some natural number k such that it terminates within k steps.

2.2 The High Level Source Language

Our source is a fairly conventional simply-typed, call-by-value functional language with recursion. The only mild novelty is that we have included a rudimentary form of refinement typing on the integer and boolean base types, indexing them by arbitrary Coq predicates:

$$A, B ::= \text{Nat } P_n \mid \text{Bool } P_b \mid A \to B \mid A \times B$$

where $P_n \subseteq \mathbb{N}$ and $P_b \subseteq \mathbb{B}$. The refinements are not very sophisticated, but their addition is easy, mildly interesting and strengthens the type soundness result somewhat. Entailment on refinement predicates induces a subtype relation $A <: B$:

$$\frac{P_n \subseteq P'_n}{\text{Nat } P_n <: \text{Nat } P'_n} \qquad \frac{P_b \subseteq P'_b}{\text{Bool } P_b <: \text{Bool } P'_b}$$

$$\frac{A <: A' \qquad B <: B'}{A \times B <: A' \times B'} \qquad \frac{A' <: A \qquad B <: B'}{A \to B <: A' \to B'}$$

The terms are given by the following grammar:

$$\begin{array}{lll} M, N, P & ::= & b \mid n \mid x \mid \lambda x.M \mid MN \mid \text{Fix} f\ x = M \mid \\ & & M \star N \mid M > N \mid \texttt{if } M \texttt{ then } N \texttt{ else } P \mid \\ & & \langle M, N \rangle \mid \pi_1(M) \mid \pi_2(M) \end{array}$$

where $b \in \mathbb{B} = \{\mathtt{t}, \mathtt{f}\}$, $n \in \mathbb{N}$ and $\star \in \{-, +, *\}$.

The type rules for our source language are shown in Figure 1. To deal with the presence of simple refinements, the type rules $[Op]$ and $[Gt]$ for the basic operations on integers involve abstract (in the sense of abstract interpretation) operations, lifting the concrete ones to predicates. The definitions of these abstract operations are:

$$\begin{array}{lll} P_1 \mathbin{\widehat{\star}} P_2 & = & \{n \mid \exists x \in P_1, \exists y \in P_2, (x \star y = n)\} \\ P_1 \mathbin{\widehat{>}} P_2 & = & \{b \mid \exists x \in P_1, \exists y \in P_2, (x > y \Leftrightarrow b)\}. \end{array}$$

The boolean refinements show up in the typing of conditionals, allowing one to type the branches under additional (meta-level) hypotheses about which way the test evaluated, much as in Hoare logic, GADTs or dependent type theories.

The Coq definition of the syntax and type rules of our source language comprises inductive definitions that translate the types, subtyping relation, untyped syntax and typing relation shown above in an entirely straightforward way. The only real differences between the LaTeX and Coq is that the Coq syntax uses de Bruijn indices rather than variable names, and that (slightly hackily) the Fix constructor in the Coq datatype really only binds f with the typing rule then insisting that the body be a lambda abstraction, binding x.

3. The Compiler

The compiler is a structurally-inductive Coq function that takes as input an untyped source expression `e` and produces two pieces of low-level code: an auxiliary section, containing the code for the bodies of functions occurring in `e`, and a main section, which actually computes the value of `e`. In more detail:

```
Fixpoint compile (e : Exp)
                 (aux_code: list instruction)
                 (aux_next : nat)
                 (alloc dealloc:nat) {struct e}
  : (  (nat -> list instruction * nat * nat)
     * list instruction * nat) := ...
```

Here `aux_code` is a list of already-generated auxiliary instructions, `aux_next` is the code address from which further auxiliary instructions should be produced, and `alloc` and `dealloc` are the entry points of the allocation and deallocation routines with which the compiled code will eventually be linked. The second and third components of the return value are the extended auxiliary code and and updated `aux_next`. The first component of the return value is a function that takes as input a start address for the main code and produces a triple comprising the instructions of the main code, the amount of stack space required by those instructions and the (slightly unnecessary) next free main code address. The generation of the main code is delayed in this way because we do not initially know how large the auxiliary code will be, and hence where we may produce the main code.

When compiled code is running, the memory is broadly divided into a number of regions:

- The low-numbered locations 0-9, used in a register like fashion for passing arguments and returning results by compiled functions and by the allocator routines, as workspace and as special registers (stack pointer etc.).
- The private storage of the memory allocator, comprising free memory and whatever private datastructures the allocator module uses to keep track of what is free.
- The language heap, storing allocated pairs, closures and environments. We sometimes call this the 'cloud'.
- A linked list of allocation records comprising the call stack, each of which includes a local stack for expression evaluation.

The important pseudo-registers are

- `arg` and `ret`, which are used in the call/return convention of the memory allocator.

$$[Var]\ \frac{}{\Gamma, x : A \vdash x : A} \qquad [Bool]\ \frac{b \in P}{\Gamma \vdash b : \text{Bool } P} \qquad [Nat]\ \frac{n \in P}{\Gamma \vdash n : \text{Nat } P}$$

$$[Abs]\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \qquad [App]\ \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

$$[Fix]\ \frac{\Gamma, f : A \to B, x : A \vdash M : B}{\Gamma \vdash \text{Fix } f\, x\ =\ M : A \to B} \qquad [If]\ \frac{\Gamma \vdash M : \text{Bool } P \quad P\mathtt{t} \Rightarrow \Gamma \vdash M_1 : A \quad P\mathtt{f} \Rightarrow \Gamma \vdash M_2 : A}{\Gamma \vdash \mathtt{if}\ M\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 : A}$$

$$[Op]\ \frac{\Gamma \vdash M : \text{Nat } P_1 \quad \Gamma \vdash N : \text{Nat } P_2}{\Gamma \vdash M \star N : \text{Nat}(P_1 \widehat{\star} P_2)} \qquad [Gt]\ \frac{\Gamma \vdash M : \text{Nat } P_1 \quad \Gamma \vdash N : \text{Nat } P_2}{\Gamma \vdash M > N : \text{Bool}(P_1 \widehat{>} P_2)}$$

$$[Pair]\ \frac{\Gamma \vdash M_i : A_i\ (i = 1, 2)}{\Gamma \vdash \langle M_1, M_2 \rangle : A_1 \times A_2} \qquad [Proj]\ \frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash \pi_i(M) : A_i\ (i = 1, 2)} \qquad [Coerce]\ \frac{\Gamma \vdash M : A \qquad A <: B}{\Gamma \vdash M : B}$$

Figure 1. Typing rules for PCF_v

- `sp`, pointing to the top of the current evaluation stack (a contiguous block of memory whose maximum depth is statically determinable, even for untyped code).
- `env`, pointing to the current environment, which is represented as a linked list of cons cells in the language heap.
- `wk`, used as temporary workspace and to pass the block to be freed to the deallocator.

We now describe the code generated for some of the high-level constructs in our language and explain the conventions and representations it uses. As an aid to understanding these descriptions, Figure 2 roughly illustrates part of a 'typical' memory layout, showing the four main regions. The dark blue gradient-filled cons cells are the ones that are part of environments; as we shall explain, these are initially allocated along with frames, but should be thought of as belonging to the language heap.

We should state explicitly at this point that there is no garbage collector, so compiled code will leak memory in the heap. We do, however, deallocate activation records when functions return.

Integer constants. To compile code for the integer constant n, we first increment the stack pointer and then store the value n on the new top of the stack:

```
 [sp]   ← [sp]+1
[[sp]]  ← n
```

In the figure, the integer 42 has been pushed and then another value (a closure) has been pushed on top of it.

Variables. Our source language represents variables by de Bruijn indices. So the code for a reference to variable n has to look up the n-th element of the environment and push it onto the top of the stack. The loop stepping through the linked list representing the environment is unrolled in place, like this:

```
 [sp]   ← [sp]+1
 [wk]   ← [env]
 [wk]   ← [[wk]+1]
 ...  n times  ...
 [wk]   ← [[wk]+1]
[[sp]]  ← [[wk]]
```

In the figure, the zeroth environment entry (the argument to the current function call) is a heap-allocated pair, the entry at position one is a recursive closure and that at position two is the same pair as at position zero.

Pairs. To construct a pair, we assume that elements of the pair have already been constructed on the top of the stack (in the 'wrong' order). We then need to call the allocator. We load the register **arg** with the size of the required block (here 2) put the return address `lab+4` into **ret** and jump to the allocation routine at address `alloc`. On return to `lab+4`, **ret** will point to a fresh block of two cells. We then (in a slightly optimized way) pop the top two elements from the stack, store them in the new block and push the address of that block:

```
lab:    [sp]   ← [sp]-1
        [arg]  ← 2
        [ret]  ← lab+4
        jmp alloc
lab+4:   [[ret]]   ← [[sp]]      // store snd
        [[ret]+1] ← [[sp]+1]    // store fst
         [[sp]]    ← [ret]       // push pair
```

In the figure, the argument to the current function is the pair `((true,3),false)`.

Closures To compile a lambda abstraction $\lambda x.M$ we compile, in the auxiliary code, the code for the body M, preceded by a header that allocates space for a new activation record and does bits of callee-saving, and followed by an epilogue that restores caller context and returns a value. Then in the main code, we build and push a closure value that pairs the current environment with a pointer to the wrapped body code.

The function header expects an argument and a pointer to the closure being called to be on the top of the caller's stack. It allocates `stack_size+5` words for the new activation record, grabs the stored environment from the closure object and stores it at offset 1 in the new activation record. The argument is then popped from the caller's stack and stored at offset zero in the new activation record. The caller's environment is stored at offset 2, and his stack pointer at offset 3. We then set the environment pointer up for the body of the function by setting it to the base of the new activation record and set the stack pointer to offset 4 within the record. Note how the first element of the new environment list (the argument and the pointer to the rest of the environment) is contiguous with the remainder of the activation record. Here is the code:

```
lab:    [arg] ← stack_size + 5 allocate frame
        [ret] ← lab + 3
        jmp alloc
lab+3:  [wk] ← [[sp]]          take closure ptr
```

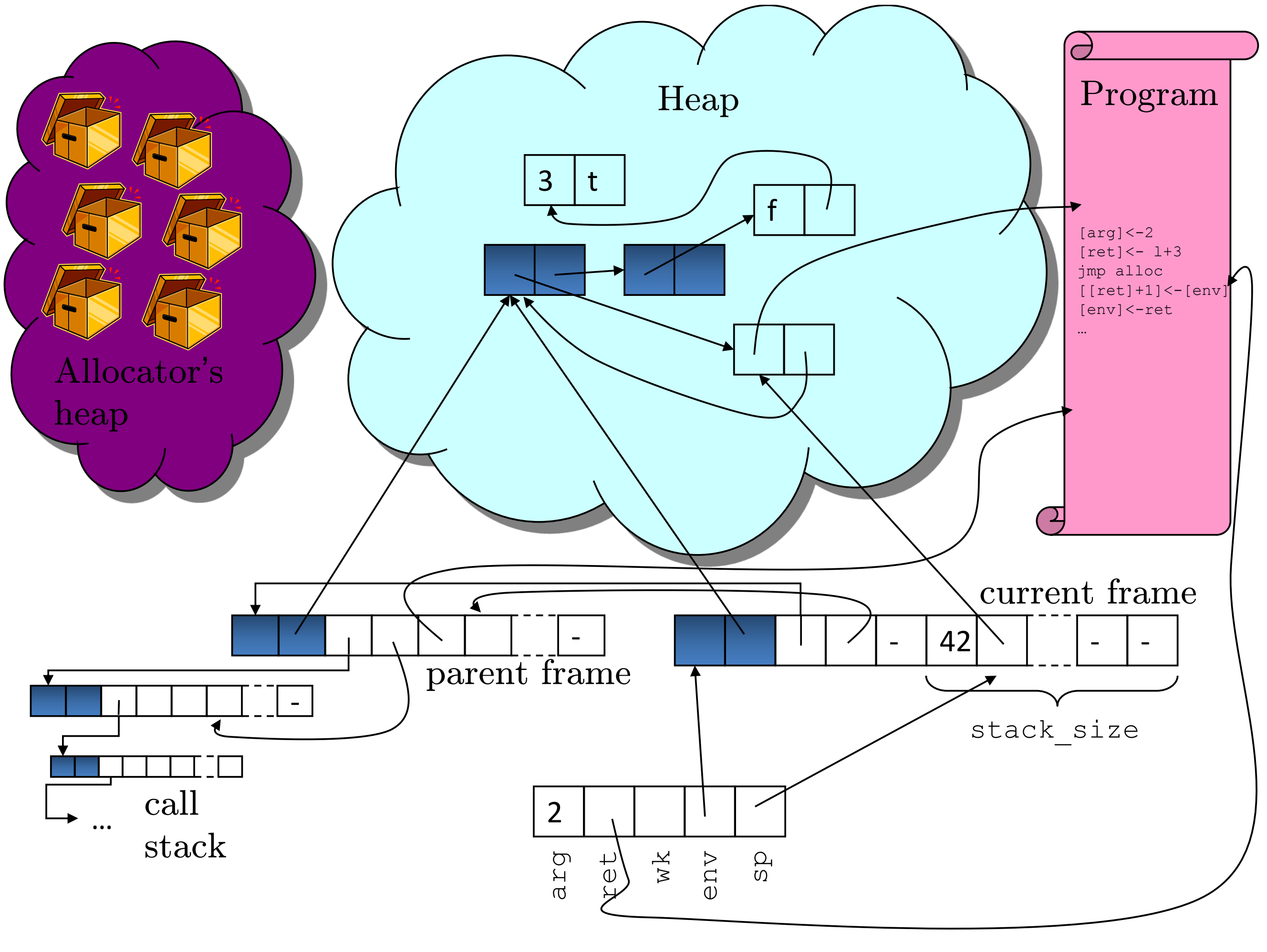

Figure 2. Memory layout at runtime

```
[[ret]+1] ← [[wk]+1]   store env ptr
[sp] ← [sp]-1          focus on argument
[[ret]] ← [[sp]]       store argument in env
[[ret]+2] ← [env]      store calling env
[[ret]+3] ← [sp]       store calling sp
[env] ← [ret]          switch to new env
[sp] ← [ret] + 4       switch to new stack
```

The saved environment and stack pointer can be seen in the figure as the two left-pointing arrows from the current frame to the parent frame. The head cell of the current environment is the darker pair of cells at the base of the current frame.

The function epilogue has the job of restoring the caller's environment and stack pointer, putting the value computed by the function on the caller's stack, deallocating the callee's frame and returning to the caller's code. The interesting points here are that we do not deallocate the first two elements of the frame that we originally allocated, as they are part of the environment and may by now be shared by other closures, and that the return address is stored at offset 4 in the *caller*'s frame:

```
lab:    [wk]   ← [[sp]]          save return value
        [sp]   ← [[env]+3]       restore old sp
       [[sp]]  ← [wk]            push return value
        [wk]   ← [env] +2        base to free
        [env] ← [[env]+2]        restore old frame
        [arg] ← stack_size+3     size to free
        [ret] ← lab + 8
        jmp dealloc
lab+8:  jmp [[env]+4]            return to caller
```

In the figure, one can see the return address to which we will jump when returning from the current call at offset 4 in the parent frame, pointing into the program.

The code to actually build a closure allocates a new pair, fills in the first field with the code pointer to the appropriate function header and the second with the current environment, and pushes the address of the pair onto the stack.

Application. The code produced for an application assumes that there is an argument and a pointer to a closure on the stack. It stores a return address in the current frame and then branches to the code pointer part of the closure:

```
lab:    [[env]+4] ← lab+3
        [wk] ← [[sp]]
        jmp [wk]
lab+3:  ...
```

Recursive closures. The compilation scheme for recursive function $\mathrm{Fix} f x = M$ relies on that non-recursive functions. We first extend the environment with an unfilled hole using the following code:

```
lab:   [arg] ← 2
       [ret] ← lab+3
       jmp alloc
lab+3: [[ret]+1] ← [env]
       [env] ← [ret]
```

We then compile code to build the closure for $\lambda x.M$ in this environment, just as above. After that, we tie the recursive knot by overwriting the hole with the address of the constructed closure and finally restore the original environment:

```
[[env]] ← [[sp]]     // knot
 [env]  ← [[env]+1] // restore
```

In the figure, the value on the top of the stack (and at position 1 in the environment) is a recursive closure. Observe that the environment part of the closure, the second component, points to the two element environment drawn within the cloud, and that the zeroth value in that environment is the closure itself.

4. Low-level Relations

As we said in the introduction, we specify low-level code in terms of binary relations, rather than the unary predicates that are more common in Hoare-style program logics. There are three main types of relation with which we work: over states, over natural numbers and over programs.

In defining relations over states, we will make much use of a form of separating conjunction, $\otimes$. Previously we worked with relations equipped with *accessibility maps* (Benton and Leperchey 2005), specifying the (state-varying) part of the state about which a particular relation cares (its *support*), and used disjointness of supports to define the separating conjunction. However, supported relations do not admit general notions of disjunction or existential quantification, which made some definitions in our previous work rather complex: we had to define combinators for special shapes of existentially quantified formula in which the satisfying witness was always uniquely determined. Furthermore, every inductive relation required separate inductive definitions for the relation and for its support *and* an inductive proof that the one supported the other, which was rather painful. In the current work, we have moved from supported relations over total states to working with relations over partial states (which is just the way everybody else working in separation logic does things). Our original motivation for eschewing partial states was that we did not wish to introduce any fictional notion of 'going wrong' into the operational semantics. We still manage to achieve that, however, by defining the operational semantics over total states (a special case of partial ones) and restricting attention to relations that are preserved by extension of partial states.

There is also an additive conjunction of relations, $\times$, which plays a larger role in giving the semantics to our functional language than was the case for an imperative one, because there are more potentially shared immutable runtime datastructures in the functional case.

We use various different relations over natural numbers, but the most interesting construction is the 'perp' operation $(\cdot)^\top$, which takes a binary relation on states, R, and produces a relation over natural numbers $R^\top$. This is the way we specify code pointers: $R^\top$ relates two natural numbers l and l' if for any states s and s' related by R, executing from l in initial state s and from l' in state s' yields *equitermination*: either both executions halt or both executions diverge. Equitermination is our basic notion of equivalence of observable behaviour, and all our program specifications will be in terms of equitermination when placed in related contexts.

Of course, we can only talk about whether or not jumping to a particular address in a particular state halts or not if we specify the program as well. For this reason, all our state relations and natural number relations are also parameterized by a pair of programs. Finally, to deal with recursion, we also index all our relations by natural number step-indices (Appel and McAllester 2001). The intuition here is that if a relation represents a particular notion of equivalence, then its k-th approximant is 'indistinguishable within k steps'. Rather than manipulate natural numbers everywhere (as we did in our previous work), we now use a modal operator (Appel et al. 2007) to tweak step-indices just where they matter. A further advantage of making this change is that we no longer build step manipulation into the definition of $(\cdot)^\top$, which means that there is now a well-behaved Galois connection between relations on states and relations on natural numbers.

We now sketch the more formal definitions of the relations and relation constructors with which we use.

4.1 Relations on states and naturals

DEFINITION 1 (State relation). *A relation in* $\mathbf{stateRel}$ *is a predicate over two partial states, two programs and a step index, satisfying a monotonicity condition with respect to decreasing indices and extension of partial states:*

```
Record stateRel : Type := mkStateRel {
  R :> pstate → pstate → program → program
       → nat → Prop;
  stateRel_cond : ∀ s1 s2 s1' s2' p1 p2 k1 k2,
   (R s1 s2 p1 p2 k1  ∧  s1 ⊂ s1'  ∧  s2 ⊂ s2'
   ∧ k2 ≤ k1)  →  R s1' s2' p1 p2 k2
}.
```

where `pstate` *denotes partial functions from* `nat` *to* `nat` *and* $s \subset s'$ *means inclusion of partial functions.* $\mathbf{stateRel}$*s are ordered by:*

```
Definition stateRelLeq (R R' : stateRel) :=
  ∀ s s' p p' k, R s s' p p' k →  R' s s' p p' k.
```

and we will write $\preceq$ *for this order.*

Disjointness of partial states allows us to define a multiplicative tensor product, or separating conjunction, of relations (Reynolds 2002; Yang 2007), which we here give in mathematical notation:

DEFINITION 2 (Separating conjunction). *The separating conjunction* $R \otimes R'$ *of two relations in* $\mathbf{stateRel}$ *is the* $\mathbf{stateRel}$ *defined by*

$$(R \otimes R')\, s\, s'\, p\, p'\, k \overset{def}{\Longleftrightarrow} \begin{array}{l} \exists s_1\ s_2\ s_1'\ s_2',\ s = s_1 \# s_2 \wedge s' = s_1' \# s_2' \\ \wedge R\, s_1\, s_1'\, p\, p'\, k \wedge R'\, s_2\, s_2'\, p\, p'\, k \end{array}$$

where $s = s_1 \# s_2$ *means that the domains of* s_1 *and* s_2 *are disjoint and that the union of* s_1 *and* s_2 *is equal to* s.

DEFINITION 3. *An element of* $\mathbf{natRel}$ *is a relation between two natural numbers, two programs and a step index, satisfying a monotonicity condition:*

```
Record natRel : Type := mkNatRel {
  R :> nat → nat → program → program
       → nat → Prop;
  natRel_cond : ∀ l l' p p' k1 k2,
                 (R l l' p p' k1  ∧  k2 ≤ k1)
                   →  R l l' p p' k2
}.
```

and again there is a natural order:

```
Definition natRelLeq (r r' : natRel) :=
  ∀ l l' p p' k, r l l' p p' k →  r' l l' p p' k.
```

We overload Top to mean the constantly total relation on both states and naturals. The following is a (non-exhaustive) list of some of the primitive $\mathbf{stateRel}$s we will combine with $\otimes$ to build specifications:

$$\begin{aligned}
&(\{n_1, n_2\} \mapsto r)\ s\,s'\,p\,p'\,k \\
\stackrel{\text{def}}{\iff}\ &\exists l, l'\ s(n_1) = l \wedge s'(n_2) = l' \wedge r\,l\,l'\,p\,p'\,k\ (r \in \mathbf{natRel})
\end{aligned}$$

$$\begin{aligned}
&(\{n_1, n_2\} \mapsto \{m_1, m_2\})\ s\,s'\,p\,p'\,k \\
\stackrel{\text{def}}{\iff}\ &s(n_1) = m_1 \wedge s'(n_2) = m_2
\end{aligned}$$

$$n \mapsto \{m_1, m_2\} \quad \stackrel{\text{def}}{=} \quad \{n_1, n_2\} \mapsto \{m_1, m_2\}$$

$$(liftP)\,s\,s'\,p\,p'\,k \quad \stackrel{\text{def}}{\iff} \quad P \quad (P : \mathtt{Prop})$$

$$\{n_1, n_2\} \mapsto - \quad \stackrel{\text{def}}{=} \quad \{n_1, n_2\} \mapsto \mathtt{Top}$$

$$\mathtt{Block}\ m\,n \quad \stackrel{\text{def}}{=} \quad \bigotimes_{0 \le i \le n-1} (m + i \mapsto -)$$

$$\begin{aligned}
&(\mathtt{Topfrom}\ m\ m')\,s\,s'\,p\,p'\,k \\
\stackrel{\text{def}}{\iff}\ &\forall n\,n', n \ge m \Rightarrow n' \ge m' \Rightarrow (\{n, n'\} \mapsto -)
\end{aligned}$$

The separating conjunction is crucial when we wish to make an update to the store whilst guaranteeing that certain other parts of the relation are not invalidated. However, we will also need to reason about the relatedness of closures, pairs and environments stored in the heap of our functional language, which are all allowed to share in unpredictable ways. To combine these specifications, we need a *non*-separating (Cartesian, additive) form of conjunction:

DEFINITION 4 (Additive conjunction). *The conjunction $R \times R'$ of two relations in $\mathbf{stateRel}$ is the relation defined by*

$$(R \times R')\,s\,s'\,p\,p'\,k \quad \stackrel{\text{def}}{\iff} \quad R\,s\,s'\,p\,p'\,k \wedge R'\,s\,s'\,p\,p'\,k.$$

PROPOSITION 1. *Viewing $\mathbf{stateRel}$ with the order $\preceq$ as a category, $\otimes$ gives a symmetric monoidal structure and $\times$ a cartesian product. These are related by the distributive law:*

$$(R \times S) \otimes T \preceq (R \otimes T) \times (S \otimes T).$$

4.2 An adjunction with $\mathbf{natRel}$

We relate relations on states and relations on code pointers via an adjunction between $\mathbf{stateRel}$ and $\mathbf{natRel}$.

DEFINITION 5 (adjunction $\mathbf{Perp} \dashv \mathbf{Perp_{nat}}$). *The two functors (order-reversing functions) $\mathbf{Perp}$ and $\mathbf{Perp_{nat}}$ defined by*

```
Definition Perp (S:stateRel) :=
 fun p p' l l' k ⇒ ∀ s s' j,
  (j ≤ k ∧ S s s' p p' j) →
  ((kstepterm j p s l → terminates p' s' l') ∧
   (kstepterm j p' s' l' → terminates p s l)).

Definition Perp_nat (L:natRel) :=
 fun s s' p p' k ⇒ ∀ l l' j,
  (j ≤ k ∧ L p p' l l' j) →
  ((kstepterm j p s l → terminates p' s' l') ∧
   (kstepterm j p' s' l' → terminates p s l)).
```

yield an adjunction (Galois connection):

$$\begin{tikzcd}
\mathbf{stateRel} \arrow[rr, bend left, "\mathbf{Perp}"] \arrow[rr, phantom, "\perp"] & & \mathbf{natRel}^{op} \arrow[ll, bend left, "\mathbf{Perp_{nat}}"]
\end{tikzcd}$$

In mathematical notation, we'll write $R^\top$ for the image of the relation R by $\mathbf{Perp}$. In our previous work, we defined Perp with a strict less than ($<$) relation on the step indices, which gave an operator without an adjoint, a point to which we will return later.

DEFINITION 6 (judgement). *We say two programs p and p' are* equidivergent*, or* equiterminate *under the relation $R \in \mathbf{stateRel}$ at points l and l' if they are in the relation $R^\top$ for all indices:*

$$\models p, p' \rhd l, l' : R^\top \ \stackrel{\text{def}}{=}\ \forall k,\ R^\top\,l\,l'\,p\,p'\,k.$$

The above form of judgement is the one that we will use in specifying the relatedness of program fragments. Instead of saying that two programs are related if they map states satisfying a prerelation into states satisfying a postrelation, we specify and reason backwards, in a 'continuation-passing style', saying that two programs are related when *if* the exit points are equidivergent under the postrelation *then* the entry points are equidivergent under the prerelation.

EXAMPLE 1. *For any two programs p and p' containing the following two instructions*

```
  [5]   ← [5] + 1
 [[5]]  ← [0]
```

at respective addresses $l, l+1$ and $l', l'+1$, and for any *$R \in \mathbf{stateRel}$, if*

$$\begin{aligned}
&\models p, p' \rhd l+2, l'+2 : \\
&\quad ((0 \mapsto \{28, 15\}) \otimes (5 \mapsto 13) \otimes (13 \mapsto -) \otimes R)^\top
\end{aligned}$$

then

$$\begin{aligned}
&\models p, p' \rhd l, l' : \\
&\quad ((0 \mapsto \{28, 15\}) \otimes (5 \mapsto 13) \otimes (13 \mapsto \{28, 15\}) \otimes R)^\top
\end{aligned}$$

This is just the 'doubling up' of a very simple unary property, but it serves to illustrate a simple use of perping and the way in which we use quantification over separated relations to make frames explicit: we do *not* have a 'frame rule' like that of separation logic, but achieve much the same effect using second-order quantification. Later, we'll see more interesting relational judgements involving non-trivial behavioural relationships between data and code pointers in the stores on the two sides. Note that specifications like this are 'partial', in that they are also satisfied by a pair of programs which always diverge (or indeed always halt) when started at the respective entry points. Relational reasoning is stronger regarding termination than traditional partial correctness in unary logics, since termination can be captured via equitermination with something that terminates, but the diagonal part of this kind of specification on code (i.e. which program fragments are related to themselves) does not rule out divergence.

4.3 Internalised quantification

We define $\mathbf{stateRel}$s quantified both universally and existentially over arbitrary Coq types in a straightforward higher-order style.

DEFINITION 7 (quantifiers). *If X is a type and $h : X \to \mathbf{stateRel}$, define*

$$\begin{aligned}
(\mathtt{Ex}\ h)\,s\,s'\,p\,p'\,k \quad &\stackrel{\text{def}}{\iff} \quad \exists x, h x\,s\,s'\,p\,p'\,k \\
(\mathtt{All}\ h)\,s\,s'\,p\,p'\,k \quad &\stackrel{\text{def}}{\iff} \quad \forall x, h x\,s\,s'\,p\,p'\,k
\end{aligned}$$

4.4 The modality and the Löb rule

So far, most of our constructions have essentially just passed step indices around. The reason they are there is to enable inductive reasoning about looping and recursion. Previously, we did this by making the index j strictly smaller than k in the definition of Perp and did explicit induction over indices to prove loops and recursion. Following the work of Appel et al. (2007), we now abstract and encapsulate some of this reasoning in a modality $\diamond$, which satisfies a form of the 'Löb rule' (sometimes also called the Gödel-Löb axiom). This modal rule is given by

$$\frac{\diamond\alpha \quad \vdash \quad \alpha}{\vdash \quad \alpha}$$

understood as *"if α is valid assuming that α is valid in the future, then α is always valid"*. Reading 'in the future' as 'after the machine has taken at least one step', this yields an induction scheme that we can use to reason about recursion.

DEFINITION 8 ($\diamond$ modality). *The modality $\diamond$ ('later') is defined on elements r of* $\mathbf{natRel}$ *by*

$$\diamond r\, l\, l'\, p\, p'\, k \quad \stackrel{def}{\Longleftrightarrow} \quad \exists j < k,\ r\, l\, l'\, p\, p'\, j$$

The modality $\diamond$ satisfies a lemma that is the essence of a continuation-passing version of the Löb rule:

LEMMA 1 (Löb rule). *For all programs p and p', all locations ptr, ptr', all program points l and l', and all* $\mathbf{stateRel}$*s R, the following rule is sound:*

$$\frac{\models p, p' \rhd l, l' : ((ptr, ptr' \mapsto \diamond((ptr, ptr' \mapsto l, l') \otimes R)^{\top}) \otimes R)^{\top}}{\models p, p' \rhd l, l' : ((ptr, ptr' \mapsto l, l') \otimes R)^{\top}}$$

5. Specifications and Verifications

In this section we explain the specifications for compiled code and for the memory allocation module on which compiled code relies.

5.1 Allocator specification

Code produced by the compiler expects to be linked with a memory allocation module, which it uses to allocate and deallocate activation records and to allocate data structures in the heap. The reasoning is properly modular: we have an independent specification of the allocator to which the specification of compiled code refers, and we independently verify the output of the compiler and particular allocator implementations. The specification of the allocator is essentially the same as in our previous work, but with a slight twist relating to termination.

We have said that we generally use specifications that do not rule out divergence. This works for interpreting types of our high-level language, since those all include divergent programs. However, were we to try to prove that two different low-level program fragments (e.g. the compilation of two different source expressions) were in the relation associated with a type, relying only on an allocator specification that allows divergence, there would be a problem: if the two fragments make different calls to the allocator, then we would not be able to prove that the programs equiterminate. So we have slightly modified the allocator specification of our earlier work to ensure totality.

DEFINITION 9 (total judgment). *We define when two programs p and p' satisfy* `total_judgment` $R_{pre}\, R_{post}\, p\, p'\, l_0\, l'_0$ *for program points l_0 and l'_0, R_{pre}* : `nat` $\to$ `nat` $\to$ $\mathbf{stateRel}$ *and R_{post}* : $\mathbf{stateRel}$ *thus*

$$\begin{array}{rl} & \texttt{total\_judgment}\ R_{pre}\, R_{post}\, p\, p'\, l_0\, l'_0 \\ \stackrel{def}{\Longleftrightarrow} & \forall l_1\, l'_1\, k\, s_0\, s'_0, (R_{pre} l_1 l'_1)\, s_0\, s'_0\, p\, p'\, k \Rightarrow \\ & \quad \exists j_1\, j'_1\, s_1\, s'_1, \quad \texttt{kstepreduce}\ j_1\ p\ s_0\ s_1\ l_0\ l_1 \\ & \qquad\qquad \wedge \quad \texttt{kstepreduce}\ j'_1\ p'\ s'_0\ s'_1\ l'_0\ l'_1 \\ & \qquad\qquad \wedge \quad R_{post}\, s_1\, s'_1\, p\, p'\, k \end{array}$$

where `kstepreduce` $j_1\ p\ s_0\ s_1\ l_0\ l_1$ *means that the program p takes j_1 steps from the configuration $\langle s_0, p, l_0\rangle$ to the configuration $\langle s_1, p, l_1\rangle$.*

Note that total judgements are more like those of traditional Hoare logics, and that this is less behavioural (i.e. more intensional) than our other form of specification: the relation R_{pre} is explicitly parameterized by the return addresses, so we are asserting that particular program counters will be reached.

We now define the prerelations and postrelations used in the specs of the three entry points of the allocator, initialization, allocation and deallocation. Each of these is parameterized by a relation R_a which will be a private invariant relating two different, equivalent, allocators. R_a will be existentially quantified (i.e. made abstract) on the outside of the whole module. Each of the other parameters will be universally quantified over the total judgement for that particular entry point.[1]

The parameterized relations `RPre_init` and `RPost_init` for the initialization routine of the allocator are given by

```
Definition RPre_init (Rₐ:stateRel) (n₅ n'₅ n₆ n'₆
l l' : nat) :=
  (ret ↦ l,l') ⊗ (sp ↦ n₅,n'₅) ⊗ (env ↦ n₆,n'₆)
  ⊗ Topfrom 10 10.
```

```
Definition RPost_init (Rₐ:stateRel) (n₅ n'₅ n₆ n'₆
: nat) := Rₐ ⊗ (ret ↦ −) ⊗ (sp ↦ n₅, n'₅)
⊗ (env ↦ n₆, n'₆).
```

```
Definition total_init Ra p p' init init' :=
 forall n5 n'5 n6 n'6, total_judgment
   (RPre_init Ra n5 n'5 n6 n'6)
   (RPost_init Ra n5 n'5 n6 n'6) p p' init init'.
```

The intuition here is that if one calls the initialization routines of two related allocators, passing return addresses l and l', and with the values n_5 and n'_5 in the pseudo-register sp on the two sides, and n_6 and n'_6 in env, then the two sides will each take some number of steps to reach l and l' in two states in which the invariant R_a has been established, the original values in sp and env have been preserved and nothing is guaranteed about the contents of ret.

The parameterized relations `RPre_alloc` and `RPost_alloc` for the allocation routine are given by

```
Definition RPre_alloc (Rₐ R꜀:stateRel) (n n' n₅ n'₅
n₆ n'₆ l l': nat) :=
  (ret ↦ l,l') ⊗ Rₐ ⊗ R꜀ ⊗ (arg ↦ n,n')
  ⊗ (sp ↦ n₅,n'₅) ⊗ (env ↦ n₆,n'₆).
```

```
Definition RPost_alloc (Rₐ R꜀:stateRel) (n n' n₅ n'₅
n₆ n'₆: nat) :=
 (Ex fb,Ex fb',((ret ↦ fb,fb') ⊗
                          (Block fb fb' n n')))
 ⊗ Rₐ ⊗ R꜀ ⊗ (arg ↦ −)
 ⊗ (sp ↦ n₅,n'₅) ⊗ (env ↦ n₆,n'₆).
```

```
Definition total_alloc Ra p p' alloc alloc' :=
 forall Rc n n' n5 n'5 n6 n'6,
   total_judgment
       (RPre_alloc Ra Rc n n' n5 n'5 n6 n'6)
       (RPost_alloc Ra Rc n n' n5 n'5 n6 n'6)
        p p' alloc alloc'.
```

These say that if one calls related allocators in states in which the private invariant R_a holds (so the two allocators have been initialized), some separate invariant relation R_c holds on other parts of the stores, passing argument values n and n', then the executions will end up at the return addresses with R_a and R_c still holding and

[1] We have slightly simplified these specifications in this presentation by leaving out parts that just mention unused pseudo-registers.

`ret` pointing to disjoint blocks of memory of sizes n and n' on the respective sides.

The relations `RPre_dealloc` and `RPost_dealloc` used to specify the deallocator are given by

```
Definition RPre_dealloc (Ra Rc:stateRel) (fb fb'
n n' n5 n'5 n6 n'6 l l': nat) :=
  (ret ↦ l,l') ⊗ (arg ↦ n,n') ⊗ (wk ↦ fb,fb')
  ⊗ Block fb fb' n n' ⊗ Ra ⊗ Rc
  ⊗ (sp ↦ n5,n'5) ⊗ (env ↦ n6,n'6).

Definition RPost_dealloc (Ra Rc:stateRel) (n5 n'5
n6 n'6: nat) :=
  (ret↦ −) ⊗ (arg ↦ −) ⊗ (wk ↦ −) ⊗
  Ra ⊗ Rc ⊗ (sp ↦ n5,n'5) ⊗ (env ↦ n6,n'6).

Definition total_dealloc Ra p p' dealloc dealloc':=
 forall Rc fb fb' n n' n5 n'5 n6 n'6,
  total_judgment
     (RPre_dealloc Ra Rc fb fb' n n' n5 n'5 n6 n'6)
     (RPost_dealloc Ra Rc n5 n'5 n6 n'6)
      p p' dealloc dealloc'.
```

Here the precondition is that there are blocks of memory of the appropriate sizes and disjoint from R_a, R_c and the registers on the two sides. The postcondition just says that R_a, R_c still hold and that the values in sp and env are preserved.

The relation between programs p and p' that says that they have allocator modules related by R_a with entry points `init`, `alloc` and `dealloc` and `init'`, `alloc'` and `dealloc'` is then just

```
Definition AllocSpec p p' Ra init init' alloc
  alloc' dealloc dealloc' :=
     (total_init Ra p p' init init')
  ∧ (total_alloc Ra p p' alloc alloc')
  ∧ (total_dealloc Ra p p' dealloc dealloc').
```

5.2 Semantics of types

We can now define the low-level specifications corresponding to our high-level types. The basic idea is that for each source type A, we define $[\![A]\!] : nat \to nat \to \mathbf{stateRel}$ such that $[\![A]\!]\ l\ l'$ relates pairs of states in which l and l' can be interpreted as pointing to equivalent values of type A. Since our values include closures, the dependency of $\mathbf{stateRel}$s on the programs really gets used, as we will need to say that the values in the heap involve code pointers with certain behaviours. Using this notion of equivalent values in the state, we define when states contain equivalent environments and evaluation stacks of particular types. Finally, we specify compiled code fragments in terms of equitermination when started in sufficiently equivalent states, and linked with sufficiently equivalence-respecting continuations.

The inductive definition of `semantics_of_types` is shown in Figure 3. Although this looks complex, it really amounts to a fairly familiar logical relational interpretation of types, but recast in a lower-level setting that naturally introduces more details.

The first clause says that two values ptr and ptr' are equal when considered at type `Int P` when they are equal natural numbers and also satisfy the predicate P, just as one would expect. The second clause says that they are equal at `Bool P` when they both represent the same boolean value according to our chosen representation of booleans (`n2b` maps zero to false and successors to true), and that boolean satisfies the predicate P.

The third clause says ptr and ptr' represent equivalent values of type $A \times B$ in respective stores s and s' just when ptr is the address of two consecutive cells in s with contents $value$ and $value_2$, ptr' is the address of a pair of cells holding $value'$ and $value'_2$ in the state s', $value$ and $value'$ are equivalent values of type B, and $value_2$ and $value'_2$ are equivalent values of type A. So values are related at a product type if they are both pairs and their components are related pointwise.

The real work is in the fourth clause, which says what it means for values to be related at a function type. A functional value will contain some code, to which we will jump when we apply the function. So relatedness of functional values is a constraint on the behaviour of that code in certain contexts. The first thing to observe is that part of that contract will be that the code is allowed to assume that the allocator has been initialized, and must promise to maintain the allocator's private invariant - that explains why the definition of the semantics of types is parameterized by the allocator's private invariant R_a. The second thing to note is that all the functional values that are constructed by code produced by our compiler will be closures: pairs of a code pointer and an environment. But recall that we are trying to come up with maximally permissive *extensional* specifications, that will be satisfied by any code that behaves like a function when we apply it. So we certainly do not want to require that functions have environments containing values of some (even hidden) source language types. In fact, our calling convention does not even require that there *is* an environment at all – the function code gets passed a pointer to the function object itself, from which it can recover an environment if it has one, but the caller does not know anything about environments. So from an external point of view, the specification of what it means for ptr and ptr' to be related elements of $A \to B$ is just that there exists *some* private invariant relation $Rprivate$ such that ptr and ptr' point to code pointers that behave in a certain way *and* $Rprivate$ actually holds now (so the functions are set up and ready to call). In the case of code produced by the compiler, that private invariant will get instantiated with assertions about the environment part of closures. Note that we have used the additive conjunction, so the private invariant is allowed to overlap/involve the memory locations ptr and ptr' themselves; this will actually happen when there is (direct or indirect) recursion.

So what are the details of the 'certain way' in which code of equivalent functions must behave? Firstly, this is where we use the 'later' modality discussed earlier, on which we rely when showing that the code produced for recursive functions satisfies the specification. The operational intuition is that for functions to be indistinguishable for k steps, it suffices for their bodies to be indistinguishable for $k-1$ steps, as testing them (i.e. applying them) involves jumping to them, which takes a step. Underneath the modality, the specification is a perp, a requirement that the two code pointers will equiterminate whenever they are jumped to in states related by the precondition `Pre_arrow`, which is parameterized by the private invariant, the two function values themselves, the invariant of the allocator and the semantics of the types A and B. At the risk of labouring the point, note that we pass in the semantic objects here, not the syntactic types, so the arrow construction works over arbitrary (appropriately parameterized) relations.

The precondition `Pre_arrow` says under what conditions the entry points of the two functions have to promise to behave equivalently for them to be regarded as representing equal functions. We are essentially translating a CPS transformed version of the standard logical relations definition, that functions are related when they take related arguments to related results. So `Pre_arrow` relates two states just when they represent calls to the two functions with $[\![A]\!]$-related arguments and $[\![B]\!]^\top$-related continuations. The tops of the two stacks are expected to point to the original function objects (from which they can get to any local state, usually an environment, they might need). Below that on the stacks are two arguments, ptr_arg and ptr_arg' which are themselves equivalent according to $[\![A]\!]$. The allocator invariant Ra is assumed to hold, disjoint from everything else. The environment registers point to

```
Fixpoint semantics_of_types (t:ExpType) (Ra:stateRel) ptr ptr' struct t :=
  match t with
  | Int P ⇒ lift (P ptr ∧ (ptr = ptr'))
  | Bool P ⇒ lift (P (n2b ptr) ∧ (n2b ptr = n2b ptr'))
  | a ∗ b  ⇒ Ex value, Ex value2, Ex value', Ex value2', (ptr,ptr'↦value,value') ×
                  (ptr+1,ptr'+1↦value2,value2') × ⟦b⟧ Ra value value' × ⟦a⟧ Ra value2 value2')
  | a ⟶ b  ⇒ Ex Rprivate,
                     (ptr,ptr' ↦ Later ( Perp (Pre_arrow Rprivate ptr ptr' Ra (⟦a⟧) (⟦b⟧))) × Rprivate)
  end
  where "'⟦' t '⟧'" := (semantics_of_types t ).


Definition Post_arrow b (Ra Rc: stateRel) Rc_cloud (n n' stack_ptr stack_ptr': nat):=
  Ex ptr_result, Ex ptr_result',
    (stack_ptr,stack_ptr' ↦ ptr_result,ptr_result') ⊗ (stack_ptr+1,stack_ptr'+1↦-) ⊗
    ((b Ra ptr_result ptr_result') × Rc_cloud) ⊗ (workreg ↦ -) ⊗ (argreg ↦ -) ⊗ (retreg ↦ -) ⊗
    Ra ⊗ Rc ⊗ (3↦-) ⊗ (4 ↦-)  ⊗ (spreg↦ stack_ptr,stack_ptr') ⊗ (envreg↦ n,n') ⊗ unused_space.

Definition Pre_arrow R_private ptr_function ptr_function' Ra a b:=
  Ex Rc, Ex Rc_cloud, Ex n, Ex n', Ex ptr_arg, Ex ptr_arg', Ex stack_ptr, Ex stack_ptr',
    (stack_ptr,stack_ptr'↦ ptr_arg,ptr_arg') ⊗ (stack_ptr+1,stack_ptr'+1↦ ptr_function,ptr_function')
    ⊗ (R_private × a Ra ptr_arg ptr_arg' × Rc_cloud) ⊗
    ((n+4,n'+4 ↦ Later (Perp (Post_arrow b Ra Rc Rc_cloud n n' stack_ptr stack_ptr'))) × Rc) ⊗
    (workreg ↦ -) ⊗ (argreg ↦ -) ⊗ (retreg ↦ -) ⊗ Ra ⊗ (3 ↦ -) ⊗ (4 ↦ -) ⊗
    (spreg↦ stack_ptr+1,stack_ptr'+1) ⊗ (envreg↦ n,n') ⊗ unused_space.
```

Figure 3. Relational semantics of types

the callers' stack frames, each of which contains a return address at offset 4. The entry point can also assume that the private invariant $Rprivate$, relating the two functions' private storage, holds at entry. In general, this invariant will be over the language heap, which contains immutable values with unspecified sharing between them, including the representation of the argument value if that is of non-primitive type. These assumptions about the language heap are captured by an additive, non separating, conjunction of $Rprivate$, the $[\![A]\!]$-relatedness of the argument and some unknown further assumptions Rc_cloud on which the continuation may be depending. There is also an unknown invariant Rc that holds over state disjoint from the language heap, the allocators state and the pseudo registers, but which may share with the return address slot of the callers' frames. This Rc will typically be instantiated with assumptions about the linked list of activation records comprising the call stack – note again that we do not explicitly model such a list-like structure at all, as each individual function call can just treat it as abstract.

The specifications for the two related return addresses, stored in the callers' frames, are once again modalized perped formulae, asking that those two return addresses behave equivalently whenever they are returned to with states that are appropriately related according to conditions that are made precise in the definition of `Post_arrow`. The return addresses can assume that the stack pointers point to equivalent returned values according to $[\![B]\!]$ and that the local frame invariant Rc_cloud from the precondition has been preserved, sharing storage with the return value. They can also assume that the allocators' private invariant has been maintained and that the frame condition on the rest of the state, Rc, has been preserved. The environments will have been put back to what they were before the call (the preservation of n and n') and the stack pointer will be one less than it was before the call (because we have popped the argument and the function object and pushed a return value).

The above specification of type-dependent relatedness of *values* in the heap of the machine is a stepping stone on our way to writing specifications for *computations*. The actual programs that manipulate and generate values are the things we ultimately want to verify. Given a context Γ and a type A, we need to define a specification for low-level programs that corresponds to producing equal results of type A when started in equal contexts of type Γ. The definition of equal contexts of type Γ is the $\mathbf{stateRel}$ that says there are two linked lists (starting at particular locations) with elements that are pairwise related by the interpretation of the corresponding type in Γ. This is a fairly straightforward induction over Γ, using the semantics of types defined above:

```
Fixpoint semantics_of_env (env:EnvType) Ra
 current current' struct env :=
  match env with
  | nil ⇒ Top
  | h :: t ⇒ Ex ptr, Ex ptr',
             Ex next, Ex next',
     ((current,current' ↦ ptr,ptr') ×
     (current+1,current'+1 ↦ next,next')
      × ⟦h⟧ Ra ptr ptr' × ⟦t⟧ Ra next next')
  end
where  "'⟦' env '⟧'" := (semantics_of_env env).
```

Note that we have used the additive conjunction again here, so elements of the environment can share heap storage with one another. Environments do not have to be terminated and are even allowed to be cyclic (directly through the link structure, or indirectly through closure values).

Compiled expressions also make use of a local evaluation stack (not to be confused with the call stack of activation records). This is separate from the language heap (cloud) and gets read and written as values are pushed and popped. The values on the stack, however, will generally be related by relations that involve the cloud and can overlap. To specify relatedness of stacks we therefore define a 'fold' operation that relates states in which there are particular sequences of values in consecutive memory locations and disjoint regions of memory in which an additive conjunction of relations, indexed by the values in the sequences, holds.[2]

```
Definition lstack_type :=
  list ((nat → nat → stateRel) * (nat * nat)).

Fixpoint stack_description (stack_list:lstack_type)
 ptr ptr' cloudrel struct stack_list :=
  match stack_list with
  | nil ⇒ cloudrel
  | pair h (pair ptr_h ptr_h') :: t ⇒
    (ptr,ptr' ↦ ptr_h,ptr_h') ⊗
    stack_description t (ptr-1) (ptr'-1)
                ((h ptr_h ptr_h') × cloudrel)
  end.
```

We then package up a description of entire related memory configurations like this:

```
Definition memory_specification Ra Rc Rc_cloud env
stack_list stack_free stack_free' stack_ptr
stack_ptr' n n2 n3 n' n2' n3':=
  (spreg↦stack_ptr,stack_ptr') ⊗
  Block (stack_ptr+1) (stack_ptr'+1)
  stack_free stack_free' ⊗ (envreg↦ n,n') ⊗
  storing_space n n2 n3 n' n2' n3' ⊗
  stack_description stack_list stack_ptr stack_ptr'
  (⟦env⟧ Ra n n' × Rc_cloud) ⊗ (workreg ↦ -)
  ⊗ (argreg ↦ -) ⊗ (retreg ↦ -) ⊗ Ra ⊗ Rc ⊗
  (3↦-) ⊗ (4↦-) ⊗ unused_space.
```

which relates two states when the 'active' parts of the stack are related according to `stack_list`, there is a Block of unused stack slots above that, the environments on the two sides are related according to the interpretation of the type environment `env` (with appropriate sharing), the allocator invariant R_a holds and an invariant R_c holds on the rest of the state. The relation `storing_space` describes the contents of the callee-saves slots in the current activation record, we also specify that the various registers point to arbitrary values.

5.3 Type Soundness

We are finally in a position to state our main result, that the compiler produces code that respects our relational interpretation of types. The theorem as stated in Coq is given in Figure 4. Although the statement looks rather complex, what it really says is not too hard to understand. The functions we have not defined, such as `extract_code_from_globalcode`, just project components from the result of compilation. We start with a source level expression `e` which has type `a` in source-level type environment `env`. We compile `e` twice, once starting from the location `start` and linking against allocation and deallocation routines at addresses `alloc` and `dealloc`, and once starting at `start'` linking against `alloc'` and `dealloc'`.

Then for any complete program `p` that extends the main and auxiliary code produced by the first compilation, and for any `p'` that extends the code produced by the second compilation, *if* `p` and `p'` have `Ra`-related memory allocation routines at the entry points we used in the respective compilations, then we get the result about the relatedness of the behaviour of the two bits of compiled code.

The judgement about the compiled code says that executing from program counter `start` in program `p` and from program counter `start'` in program `p'` yields equitermination provided that (a) the two initial states are related by a memory specification corresponding to the type environment `env`, together with some arbitrary other bits, and (b) the labels *after* the code compiled from `e` on the two sides always equiterminate when they are started in states that are related by the same memory specification that we assumed at the entry points, modified to add $[\![A]\!]$-related values on top of the evaluation stack (with appropriate modifications to both the stack pointers and the size of the unused stack locations).

So what does that tell us? One consequence is about the behaviour of closed programs of ground type, such as `Int P`. The theorem says that if you compile such a program and link it against a well-behaved allocator then it will either always diverge or produce the same ground value, which will moreover satisfy the predicate refinement `P`. The observable behaviour will not change according to what locations are returned by the allocator, what the initial contents of any bits of memory are, where we initially put the stack, or anything else it should not depend upon. Moreover, the computation will not write to any areas of memory that it should not (because of the preservation of an arbitrary `Rc`).

More importantly, the specification is entirely modular and semantic. The proof obligations for writing non-standard implementations of higher-order functions that are extensionally indistinguishable from (and so can interoperate with) those produced by our compiler, yet might be implemented quite differently, are made explicit and could be verified without the compiler source.

6. Remarks on the Coq formalization

Our Coq formalization covers everything discussed here: the low level machine, high level language, the compiler, two allocator modules, the general relational reasoning framework, the specifications and the proof of semantic type preservation. (The famous Knuth quote "Beware of bugs in the above code; I have only proved it correct, not tried it." almost applies, though we have compiled and executed just a few simple programs within Coq and obtained the right answers.)

The entire development is around 14000 lines and relies heavily on our previous work, though few things have been left completely unchanged. The code for the memory allocators is the same, but moving to relations over partial states instead of using accessibility maps and changing the treatment of step indices have meant small changes throughout. In general, the proof assistant helps rather than hinders such evolution – one can change basic definitions and then update the scripts so that proofs of basic properties still go through surprisingly quickly and sometimes almost automatically.

We make heavy use of Setoid rewriting modulo the preorder $\preceq$, but have replace our earlier handcrafted reflective tactics for reorganising long sequences of $\otimes$-ed $\mathbf{stateRel}$s modulo associativity and commutativity with uses of the library-provided `ring` tactic. This removes the two-level (syntactic and semantic) structure of our explicitly reflective approach which we were never sufficiently disciplined to use cleanly before.

The proofs also make rather more use of specialized tactics than our earlier ones, and we have overall managed to keep the size of this formulation about the same as that of our previous one, for a simple imperative language with no procedures, despite the fact that the specifications and proofs here are much more complex and the compiler is considerably larger.

The general pattern of reasoning is, as in our earlier work, forward Hoare style proving, using rules for entailments on $\mathbf{stateRel}$s and judgements to set the goal up in the right form to apply instruction-specific lemmas such as the following, for an unconditional indirect branch:

LEMMA 2 (Branch). *For all p, p', l, l', m and R, if*

$$p(l) = \texttt{jmp [m]}, \quad p'(l') = \texttt{jmp [m]}$$

[2] This is essentially the same as the *pexconj* construction of Benton and Zarfaty (2007).

```
Theorem compiler_sound :
 forall base base' Ra init init' alloc alloc' dealloc dealloc' Γ a e (t:Γ ⊢ e : a)
       Rc Rc_cloud start start' stack_ptr stack_ptr' n n2 n3 n' n2' n3' p p' stack_list,
  let global_code := compile e nil base alloc dealloc in
  let code := extract_code_from_globalcode global_code start in
  let stack_free := extract_stacksize_from_code code in
  let global_code' := compile e nil base' alloc' dealloc' in
  let code' := extract_code_from_globalcode global_code' start' in
  let stack_free' := extract_stacksize_from_code code' in
    prog_extends_auxcode global_code p base →
    prog_extends_code global_code p start →
    prog_extends_auxcode global_code' p' base' →
    prog_extends_code global_code' p' start' →
    AllocSpec p p' Ra init init' alloc alloc' dealloc dealloc'  →
    (forall ptr ptr', ⊨ p p' ▷ (start+length (instruct_of code)) (start'+length (instruct_of code')) :
        (memory_specification Ra Rc Rc_cloud Γ ((⟦a⟧ Ra,(ptr,ptr')) :: stack_list)
            (stack_free-1) (stack_free'-1) (stack_ptr+1) (stack_ptr'+1) n n2 n3 n' n2' n3')⊤)  →
      ⊨ p p' ▷ start start' : (memory_specification Ra Rc Rc_cloud Γ stack_list stack_free stack_free'
            stack_ptr stack_ptr' n n2 n3 n' n2' n3')⊤.
```

Figure 4. Semantic Type Soundness

and

$$R \preceq (m \mapsto \Diamond R^{\top})$$

then

$$\models p, p' \rhd l, l' : R^{\top}$$

which says that we get equitermination by executing the jump instructions at l and l' in states satisfying R if R entails that the location m through which we jump points to code pointers that later yield equitermination when jumped to in states satisfying R.

The crucial lemma for proving the recursive functions is the following, proved by appeal to the Löb rule we gave earlier:

```
Lemma recursion_continuation :
forall ptr ptr' fcode fcode' Ra a b R p p',
 judgment (Pre_arrow ((⟦a⟶b⟧) Ra ptr ptr' × R)
   ptr ptr' Ra (⟦a⟧) (⟦b⟧)) p p' fcode fcode'
 →
 judgment (Pre_arrow((ptr,ptr'↦ fcode, fcode')×R)
   ptr ptr' Ra (⟦a⟧) (⟦b⟧)) p p' fcode fcode'.
```

The antecedent is what we initially prove about the code pointers `fcode` and `fcode'`: that they will equiterminate appropriately provided that `ptr` and `ptr'` (which will be the heads of the two closure environments) point to equivalent functions. The consequent is what want to conclude about the result of tying the knot: that we then get equitermination when `ptr` and `ptr'` point, respectively, to `fcode` and `fcode'` themselves.

7. Discussion

We have shown how to specify and verify a low-level relational interpretation of functional types. The construction involves a number of familiar ideas: the logical interpretation of types as relations, separation logic, orthogonality, step-indexing and so on, but putting them all together in the right way is far from trivial, and we certainly would not claim to have completely solved the problem.

The existentially-quantified private invariant $Rprivate$ at the top level of our interpretation of function types is a semantic generalization of the well-known use of existentially quantified types to abstract the type of environments in typed closure conversion (Minamide et al. 1996; Glew 1999; Ahmed and Blume 2008). Our general use of second order existential quantification over $\mathbf{stateRel}$s to express private invariants generalizes the standard treatment of type abstraction via existential types (Mitchell and Plotkin 1988).

The use of 'orthogonality' or 'perping' in realizability was pioneered by Pitts and Stark (1998) and by Krivine, and has received much attention, notably by Vouillon and Melliès (2004). A general theory of such 'tensorial negations' is investigated in more detail in the thesis of the second author (Tabareau 2008), but there are still open questions about their use in low-level settings. One such question is whether, now we have an adjunction and hence a $(\cdot)^{\top\top}$ closure operator on $\mathbf{stateRel}$s, we really need explicit step indices too; in higher-level models, biorthogonal closed relations are automatically admissible. Another very important question is how one would adapt the kinds of specification used here to real machines with finite memory; it is not obvious what the 'corresponding' theorems should be.

Note that our interpretation of function types and general computations in context constrains them to be rather pure – the extent to which they can read or write the state is severely constrained. Our hope is to be able to do equational reasoning at the level of low-level code, proving that *different* bits of machine code are in the relation associated with a particular type. However, we have not yet managed to make this work, despite our move to total correctness specifications of the allocator. The CPS treatment of prerelations and postrelations seems to give us an equational theory which is roughly that of call-by-value CPS-transformed high-level functions. Whilst this is a strong constraint on the behaviour of machine code programs, it is still too weak to prove something as trivial as the high-level commutativity of addition, as the CPS transforms of

```
let x = M in let y = N in x+y
```

and

```
let y = N in let x = M in x+y
```

are not generally observationally equivalent. A related annoyance is that our CPS interpretation does not seem to validate the well-known rule for conjunctive types

$$(A \to B_1) \wedge (A \to B_2) <: (A \to B_1 \wedge B_2)$$

or the morally equivalent rule for introducing universal quantification over logical variables used in our refinement types:

$$[\forall I]\ \frac{\Gamma \vdash M : A(i) \qquad i \notin \Gamma}{\Gamma \vdash M : \forall i. A(i)}$$

It is well known that these rules can be unsound in the presence of effects (Davies and Pfenning 2000), but our language is sufficiently pure that we should not require a value restriction here: we just have not quite captured that purity in our specifications.

Yet another wart on our otherwise beautiful theory is the way the treatment of total correctness makes explicit reference to particular code pointers. This is just about bearable for simple first-order procedures like the allocator, but would be untenable and insufficiently modular if we were to, say, try to interpret types of a normalizing lambda calculus. We are currently investigating some ideas about parametricity in the notion of 'observation' (here it was always equitermination) with respect to which one takes perps, which may help here.

One of our initial goals was that our interpretations of types be expressed in a logic that was essentially independent of the source. This seems desirable in multilanguage PCC settings, but certainly introduces some complexity. An alternative approach would be to define a 'represents' logical relation between a high-level semantics for the source language and the low-level compiled code (this would then induce a partial equivalence relation on low-level programs). Chlipala (2007) has already done this for a simple total functional language, which can be given a straightforward type-theoretic denotational semantics; we plan to do something similar using a formalized domain-theoretic semantics for our partial language. Our hope is that this will help us better understand the shortcomings of the low-level relations.

Finally, of course, we want to look at various forms of compiler correctness for high-level languages with effects. We and others have successfully applied essentially the same mathematical machinery to reasoning about higher-order functions with encapsulated local state (Ahmed 2004; Benton and Leperchey 2005; Pitts and Stark 1998; Ahmed et al. 2009), so moving those ideas down to the low level looks very doable.

There is a great deal of related work on program logics, compiler correctness and the semantics of types, much of which we have plundered here. Formal verification of compilers goes back over four decades (McCarthy and Painter 1967; Dave 2003) and has recently received increased attention, with notable automated efforts including Leroy's verification of an optimizing compiler for a C-like language (Leroy 2006). Appel's Foundational Proof Carrying Code project (Appel 2001) at Princeton has very similar goals to this work, and many of the techniques we use were introduced by Appel and his coauthors. We mention in particular the use of step-indexing (Appel and McAllester 2001; Tan et al. 2004) and its modal refinement (Appel et al. 2007). Shao's group at Yale have also done impressive work on formalizing and verifying specifications for low-level code in Coq, including memory managers, garbage collectors and other challenging pieces of systems code (Ni and Shao 2006; Yu et al. 2004). It will be interesting to see if we can do similar things in our relational style.

Acknowledgments

Many thanks to Ivana Mijajlovic, who wrote the 'non-trivial' memory allocator module and did the initial correctness proof for it, and to Uri Zarfaty, who wrote the compiler itself.

References

A. Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, Princeton, NJ, USA, 2004.

A. Ahmed and M. Blume. Typed closure conversion preserves observational equivalence. In *Proc. 13th ACM International Conference on Functional Programming (ICFP '08)*. ACM, 2008.

A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *Proc. 36th ACM Symposium on Principles of Programming Languages (POPL '09)*, 2009.

A. Appel. Foundational proof-carrying code. In *Proc. 16th IEEE Symposium on Logic in Computer Science (LICS)*, 2001.

A. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(5), 2001.

A.W. Appel, P.-A. Melliès, C.D. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. *Proc. 34th ACM Symposium on Principles of Programming Languages (POPL '07)*, pages 109–122, 2007.

N. Benton. Abstracting allocation: The new new thing. In *CSL '06*, volume 4207 of *LNCS*. Springer-Verlag, September 2006.

N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In *Proc. 7th International Conference on Typed Lambda Calculi and Applications (TLCA)*, volume 3461 of *Lecture Notes in Computer Science*, 2005.

N. Benton and U. Zarfaty. Formalizing and verifying semantic type soundness of a simple compiler. In *Proc. 9th ACM International Conference on Principles and Practice of Declarative Programming (PPDP '07)*, pages 1–12, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-769-8. doi: http://doi.acm.org/10.1145/1273920.1273922.

A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Proc. 2007 ACM Conference on Programming Language Design and Implementation (PLDI '07)*, 2007.

M. Dave. Compiler verification: a bibliography. *ACM SIGSOFT Software Engineering Notes*, 28(6), 2003.

R. Davies and F. Pfenning. Intersection types and computational effects. In *Proc. 5th ACM International Conference on Functional Programming (ICFP '00)*, 2000.

N. Glew. Object closure conversion. In *3rd International Workshop on Higher Order Operational Techniques in Semantics*, 1999.

X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proc. 33rd ACM Symposium on Principles of Programming Languages (POPL '06)*, 2006.

J. McCarthy and J. Painter. Correctness of a Compiler for Arithmetic Expressions. *Proceedings Symposium in Applied Mathematics*, 19:33–41, 1967.

Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Proc. 23rd ACM Symposium on Principles of Programming Languages (POPL '96)*, pages 271–283, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3. doi: http://doi.acm.org/10.1145/237721.237791.

J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3), 1988.

Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Proc. 33rd ACM Symposium on Principles of Programming Languages (POPL)*, 2006.

A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*. CUP, 1998.

J.C. Reynolds. Separation logic: A logic for shared mutable data structures. *17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 55–74, 2002.

N. Tabareau. *Modalités de Ressource et Contrôle en Logique Tensorielle*. PhD thesis, Université Paris Diderot (Paris 7), 2008.

G. Tan, A. Appel, K. Swadi, and D. Wu. Construction of a semantic model for a typed assembly language. In *Proc. 5th Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '04)*, 2004.

J. Vouillon and P.-A. Melliès. Semantic types: A fresh look at the ideal model for types. In *Proc. 31st ACM Symposium on Principles of Programming Languages (POPL)*, 2004.

H. Yang. Relational separation logic. *Theoretical Computer Science*, 375 (1-3), 2007.

D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for PCC: Dynamic storage allocation. *Science of Computer Programming*, 50, 2004.