# Formalizing and Verifying Semantic Type Soundness of a Simple Compiler

Nick Benton
Microsoft Research, Cambridge
nick@microsoft.com

Uri Zarfaty
Imperial College, London
udz@doc.ic.ac.uk

## Abstract

We describe a semantic type soundness result, formalized in the Coq proof assistant, for a compiler from a simple imperative language with heap-allocated data into an idealized assembly language. Types in the high-level language are interpreted as binary relations, built using both second-order quantification and a form of separation structure, over stores and code pointers 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

The last decade has seen an explosion of research into type systems, formal verification and certification for low-level code, ignited by the original papers on typed assembly language [21] and proof-carrying code [23], and fanned by the development of separation logic [29]. These developments have been driven by various forces: partly by need (as well as the traditional arguments in favour of some level of formal verification as a way to develop software that actually works, the internet has made checkable safety of mobile code more than a purely academic problem); partly by improvements in the technology of theorem provers and model checkers; and partly by the realization that conservative techniques for verifying comparatively simple properties, such as forms of memory safety, can be much easier and more efficient to apply than complete methods for showing full functional correctness, whilst still offering useful real-world guarantees. Another driving force has been the (occasionally surprising) discovery that logical, type theoretic and semantic ideas that were originally developed in fairly abstract settings, or for very high-level programming languages, are actually applicable to realistic, low-level, 'dirty' languages and systems.

In the present paper, we will be concerned with *certified compilation*, proving once and for all that a compiler always produces object code that satisfies some policy, which in this case will be type safety.

But what do we mean by type safety? For high-level languages, there are two main approaches to formalizing type soundness properties: *syntactic* and *semantic*. The difference between the two is *not* (merely) one of proof technique; they are different kinds of result.

In the syntactic approach [31], one defines a small-step operational semantics that gets stuck (makes no transition) in configurations that are considered to be bad. One then shows 'preservation and progress' – that every typeable configuration is either properly terminal or makes a transition into another typeable configuration – and can then deduce that well-typed programs don't get stuck. Syntactic type soundness is often fairly straightforward to establish, but is a rather weak and fragile result. Firstly, it is closely tied to the particular set of syntactic rules that define the type system. There is no notion of the meaning of a type $A$ as a property of phrases beyond 'being assignable the type $A$ using this particular set of rules', so there is no real notion of what it is that the types are supposed to ensure. Secondly, the introduction of stuckness into the operational semantics is something of a sleight of hand, changing the original problem to match the solution. For a simple type system and a high-level semantics, it seems reasonable to work with syntax for (untyped) phrases that explicitly distinguishes, say, functions from integer constants, or even booleans and integers, and which gets stuck when one tries to apply an integer or increment a boolean. But this becomes less tenable when the type system is intended to track more interesting properties, such as the use of locks or the reading and writing of particular parts of the store. Formulating a syntactic type soundness result then involves further changes to the operational semantics, to track extra information and possibly add new stuck states. And the more sophisticated the analysis, the more complex the instrumentation becomes. Furthermore, the notion of error is not preserved by compilation: machine code does not inherently distinguish code pointers, heap pointers, integers or booleans; no fault is raised by performing arithmetic on addresses to which one subsequently jumps or stores,[1] and compiled code, particularly when optimized, often depends upon such possibilities.

*PPDP'07* July 14–16, 2007, Wroclaw, Poland.
Copyright © 2007 ACM 978-1-59593-769-8/07/0007. . . $5.00

[1] This is, of course, not strictly true. Operating systems use memory management hardware to trap 'illegal' pointer dereferencing or jumps to addresses in pages marked as 'no execute', floats are passed in special registers, etc. But faults in compiled code certainly do not correspond exactly to errors in a high-level semantics, and a major goal of static verification should surely be to remove the need for such crude and expensive dynamic checks.

It is, of course, possible to mitigate the effect of instrumentation by also proving an erasure theorem. Even then, however, the theorem about low-level code is tied to the syntactic definition of the high-level language and its type rules. This is a significant shortcoming: compiled code nearly always relies upon a runtime system and library routines that are written directly in a low-level language, and we would also like to be able to link soundly with code compiled from other high-level languages. Without an independent low-level characterization of the intended behavioural properties of code compiled from phrases with a particular high-level type, the implementer of a library function or support routine written in C or assembler does not know what specification his code should meet in order to interoperate properly with the output of the compiler.

The semantic approach to type safety, by contrast, gives a meaning to each type that is independent of any particular set of rules for assigning those types to program phrases. The meaning of a type will be (roughly) a set of values with some property; for a given language and set of types, there can be many different analyses, of varying degrees of precision, for soundly assigning types to terms.

Interpretations of types as predicates over some untyped model of computation have a long history. Particularly relevant for us is the work of Appel and his collaborators [6, 5, 7, 30, 3] on Foundational Proof Carrying Code (FPCC). The idea of FPCC is to give a semantics to high-level types as low-level specifications expressed in a general program logic. This low-level logic is not tied to any language or type system, and proofs that a particular piece of low-level code satisfies such a specification can be generated or checked independently from any particular compilation scheme. Although the concept of FPCC is clearly parametric in just what safety property one wishes to prove and check, the only instance that has really been studied and implemented so far is *memory safety*: ensuring that 'illegal' accesses to memory cannot occur. The intensional notion of which accesses are legal is formalized by making the operational semantics of the low-level machine get stuck when certain locations are dereferenced, just as in a syntactic approach.

Memory safety is undeniably important, but is not the same as type safety. Program fragments that satisfy an interpretation of a type in the style of previous work on FPCC, whilst memory safe, can easily fail to have other rather basic properties one would expect, and on which security and compiler correctness can depend. For an ML-like source language, for example, a machine code function that simply returns its argument will be in the interpretation of the type $(\text{int} \rightarrow \text{int}) \rightarrow \text{int}$, since if one passes in the address of some closure, one will get back something that looks like an integer. But allowing the identity function to be given that type, whilst not leading to illegal memory accesses, would invalidate very basic reasoning principles for ML programs that are used by both programmers and compilers: not only are static transformations such as common subexpression elimination no longer behaviour-preserving, but the observable results of a particular compiled binary can vary according to where it is loaded in memory, the behaviour of the allocator, etc. Such possibilities violate most language-based encapsulation or security properties one can think of.

In the present paper, we work with a semantic interpretation of high-level types that uses binary relations, rather than unary predicates, over low-level code and data. One should think of these relations as carving out both a set of values *and* a type-specific notion of equality on that set of values; these are defined together because which values are judged to be in the set associated with some compound type will depend on both the sets of values and the equality relations associated with its components. The crucial case is that for functions: two values $f$ and $f'$ are in the relation interpreting $A \rightarrow B$ iff for any $x$ and $x'$ that are related by the interpretation of $A$, $f\,x$ and $f'\,x'$ are related by the interpretation of $B$. The set of values having a particular type is given by the diagonal part of the associated relation, so $f$ has type $A \rightarrow B$ just when $f$ is related to itself by the interpretation of $A \rightarrow B$; this is the usual notion of 'logical' relation [26]. Interpreting types as (partial equivalence) relations over an untyped model of computation also has a long history, but previous work has generally taken the untyped model either to be rather high-level and abstract (e.g. a domain theoretic model of the untyped lambda calculus) or low-level but with uninteresting fine structure (e.g. Gödel numbers for partial recursive functions). The difference here is that we work with a low-level, untyped model in whose structure we most certainly are interested, *viz.* machine code (albeit very idealized), and we work with a translation into that model that is representative of realistic compilation schemes (albeit for a rather toy language).

The main contribution of the present paper is not so much the actual type soundness result, but rather its general form and the methodology used for proving it. We build on our earlier work on modular specification and verification of a simple memory allocator [10]. The results have been formalized and checked in the Coq proof assistant and most of the formal parts of the present paper are presented as extracts from the proof script, using Coq syntax. Some further discussion may be found in a companion tech report [13].

## 2. Low-Level Target Machine

We work with the same straightforward operational semantics for an idealized assembly language that we used in our earlier work on allocation [10]. There is a single datatype, the natural numbers, though different instructions treat elements of that type as code pointers, heap addresses, integers, etc. The heap is a total function from naturals to naturals and the code heap is a total function from naturals to instructions. 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.

The instruction set of the machine is given by Coq inductive definitions for lvalues (*dest*), rvalues (*src*) and instructions (*instruction*). Destinations are either immediate (a fixed memory location), indirect or indirect with a fixed offset. Sources are literal values, immediate (the contents of a fixed memory location), indirect or indirect with an offset.

```
Inductive dest : Set :=
  | d_imm : nat → dest     | d_ind : nat → dest
  | d_indo : nat → nat → dest.
Inductive src : Set :=
  | s_cst : nat → src      | s_imm : nat → src
  | s_ind : nat → src
  | s_indo : nat → nat → src.
Inductive instruction : Set :=
  | i_halt : instruction
  | i_move : dest → src → instruction
  | i_add : dest → src → src → instruction
  | i_sub : dest → src → src → instruction
  | i_mult : dest → src → src → instruction
  | i_branch : src → instruction
  | i_brz : src → src → instruction
  | i_brnz : src → src → instruction.
```

The mutable heap of our machine is a function from naturals to naturals, which we represent using a record type with a single field and an implicit coercion to $(nat \rightarrow nat)$:

```
Record state : Set := State { fun_of_state :> nat → nat }.
```

```
Definition update (s:state) (n:nat) (v:nat) : state :=
  State (fun m ⇒ if beq_nat n m then v else s m).
```

We now give the meaning of sources, destinations, and the single-step semantics of instructions themselves. The latter is of an *option* type: either *None*, indicating termination, or *Some*$(s', pc')$, giving a new heap and a new program counter:

```
Definition sem_dest (de:dest) (s:state) :=
match de with
  | d_imm n ⇒ n
  | d_ind n ⇒ s n
  | d_indo ofs n ⇒ s n + ofs
end.
```

```
Definition sem_src (sr:src) (s:state) :=
match sr with
  | s_cst n ⇒ n
  | s_imm n ⇒ s n
  | s_ind n ⇒ s (s n)
  | s_indo ofs n ⇒ s (s n + ofs)
end.
```

```
Definition sem_instr (ins:instruction) (s:state) (pc:nat) :
     option (state × nat) :=
match ins with
  | i_halt ⇒ None
  | i_move de sr ⇒
  Some (update s (sem_dest de s) (sem_src sr s), S pc)
  | i_add de sr1 sr2 ⇒
    Some (update s (sem_dest de s) ((sem_src sr1 s) + (sem_src sr2
s)), S pc)
  | i_sub de sr1 sr2 ⇒
    Some (update s (sem_dest de s) ((sem_src sr1 s) - (sem_src sr2
s)), S pc)
  | i_mult de sr1 sr2 ⇒
    Some (update s (sem_dest de s) ((sem_src sr1 s) × (sem_src sr2
s)), S pc)
  | i_branch sr ⇒ Some (s, sem_src sr s)
  | i_brz srscrut srtarg ⇒
  Some (s, match sem_src srscrut s
                with 0 ⇒ sem_src srtarg s | S _ ⇒ S pc end)
  | i_brnz srscrut srtarg ⇒
  Some (s, match sem_src srscrut s
                with 0 ⇒ S pc | S _ ⇒ sem_src srtarg s end)
end.
```

A program is simply a total function from labels (naturals) to instructions, whilst a program fragment is a partial function from labels to instructions:

```
Definition program : Set := nat → instruction.
Definition program_frag : Set := nat → option instruction.
Definition program_extends_frag (p:program) (pf:program_frag)
     := ∀ n, match pf n with None ⇒ True
                                        | Some i ⇒ (p n = i) end.
```

We now define *kstepterm*, saying when a configuration comprising a program $p$, a heap $s$, and a program counter $l$ terminates in $k$ steps (the termination guarantee {*struct k*} tells Coq that the function is structurally recursive on $k$). The *terminates* predicate then holds of configurations that terminate in some number of steps:

```
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.
```

The major idealizations compared with a real machine are that we have arbitrary-sized natural numbers as a primitive type, rather than fixed-length words, and that we have separated code and data memory. Note also that there are no registers; we will simply adopt a convention of using some low-numbered memory locations in a register-like fashion.

## 3. Source Language

The source language is that of `while`-commands with natural, boolean and pair-valued expressions and a type system that supports 'strong updates'. The syntax and type rules of the language are shown in Figure 1. Expressions are typed in the context of a typing $\Gamma$ for the variables. Commands, which may update variables with values of different types, are given both a pretyping and a post-typing, recording their assumptions and effects on the store.

The Coq translation of Figure 1 is fairly direct. We use natural numbers instead of names for variables and have made elements of *EnvType*, representing store types, be total functions on the naturals (we will pass a size as well later on). Note the use of simple dependent typing for expressions and commands: *Exp env t* is the type of expressions that have type *t* in store environment *env*, and similarly for commands.

```
Inductive ExpType : Set :=
   | TInt : ExpType
   | TBool : ExpType
   | TPair : ExpType → ExpType → ExpType.
```

```
Notation "a ** b" := (TPair a b) (at level 55).
```

```
Definition EnvType := nat → ExpType.
```

```
Definition envupdate (env:EnvType) m a :=
   (fun n ⇒ if beq_nat n m then a else env n).
```

```
Inductive Exp : EnvType → ExpType → Set :=
   | EInt : ∀ env, nat → Exp env TInt
   | EBool : ∀ env, bool → Exp env TBool
   | EId : ∀ env m a (h : env m = a), Exp env a
   | EAdd : ∀ env, Exp env TInt → Exp env TInt → Exp env TInt
   | EGt : ∀ env, Exp env TInt → Exp env TInt → Exp env TBool
   | EPair : ∀ env a b, Exp env a → Exp env b → Exp env (TPair a
b)
   | EFst : ∀ env a b, Exp env (TPair a b) → Exp env a
   | ESnd : ∀ env a b, Exp env (TPair a b) → Exp env b.
```

```
Inductive Command : EnvType → EnvType → Set :=
   | CAssign : ∀ env m a, Exp env a → Command env (envupdate
env m a)
   | CSeq : ∀ env1 env2 env3, Command env1 env2 → Command
env2 env3 → Command env1 env3
   | CIf : ∀ env1 env2, Exp env1 TBool → Command env1 env2 →
Command env1 env2 → Command env1 env2
   | CWhile : ∀ env, Exp env TBool → Command env env →
Command env env.
```

## 4. Compilation

The compiler comprises a pair of functions traversing expressions and commands in the high-level language to produce lists of low-level instructions. The correctness of the generated code relies on it being linked with a memory allocator module satisfying the specification given in our previous work [10]. We call the allocator to get a statically fixed-size block for storing variables and an evaluation stack and for dynamically allocating the storage for values of pair types. Data structures generated by programs in our

Expression types $\quad A \quad := \quad \text{int} \mid \text{bool} \mid A \times A'$

Store types $\quad \Gamma \quad := \quad v_1 : A_1, \ldots, v_n : A_n$

Expressions

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}} \qquad \frac{}{\Gamma \vdash \text{false} : \text{bool}} \qquad \frac{}{\Gamma \vdash \underline{n} : \text{int}} \qquad \frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}$$

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 > e_2 : \text{bool}} \qquad \frac{\Gamma \vdash e_1 : A_1 \quad \Gamma \vdash e_2 : A_2}{\Gamma \vdash (e_1, e_2) : A_1 \times A_2} \qquad \frac{\Gamma \vdash e : A_1 \times A_2}{\Gamma \vdash \pi_1 e : A_1} \qquad \frac{\Gamma \vdash e : A_1 \times A_2}{\Gamma \vdash \pi_2 e : A_2}$$

Commands

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma, x : A \vdash (x := e) : \Gamma, x : B} \qquad \frac{\Gamma \vdash C_1 : \Gamma' \quad \Gamma' \vdash C_2 : \Gamma''}{\Gamma \vdash C_1 \,;\, C_2 : \Gamma''}$$

$$\frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash C_1 : \Gamma' \quad \Gamma \vdash C_2 : \Gamma'}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ C_1\ \texttt{else}\ C_2 : \Gamma'} \qquad \frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash C : \Gamma}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ C : \Gamma}$$

**Figure 1.** The While Language

language can involve sharing, which complicates their reclamation. We have not yet proved either garbage collection or any static memory management scheme, so for now just let the compiled code leak memory.

We adopt a convention of using memory locations 0 to 9 in a register-like fashion. The calling convention for the memory allocator is that a return address is passed in location 0 (*retreg*) and the size of the block requested is passed in location 1 (*argreg*); a pointer to the free block is returned in location 0. Code compiled from phrases of our language relies on location 6 (*envreg*) holding a pointer to the base of a block of memory, the first part of which is used to store the global variables and the remainder of which is used as a stack during the evaluation of expressions. Location 5 (*spreg*) points to the next free stack slot. Figure 2 shows a typical layout of the store at run-time. We re-emphasize that the store is *really* just a function from naturals to naturals: the intended interpretation of some of them as pointers, booleans, etc. as shown in the figure is just what we are going to formalize by giving a semantics to types.

The code for pushing a natural number *n* onto the stack does an indirect store of the constant *n* to the memory location pointed to by *spreg* and increment *spreg*:

Definition *int_code n* :=
  (*i_move* (*d_ind spreg*) (*s_cst n*)) ::
  (*i_add* (*d_imm spreg*) (*s_imm spreg*) (*s_cst* 1)) ::
  *nil*.

The code for boolean constants is similar, pushing 1 for true and 0 for false. The value of a variable *n* is obtained by indirection through *envreg* with an offset:

Definition *id_code n* :=
  (*i_move* (*d_ind spreg*) (*s_indo n envreg*)) ::
  (*i_add* (*d_imm spreg*) (*s_imm spreg*) (*s_cst* 1)) ::
  *nil*.

The sequence for the greater-than test uses subtraction. We are working with natural numbers and a subtraction operator that yields zero when the result would otherwise be negative, thus we either leave zero (representing false) or some strictly positive value, all of which we take to represent true. This encoding, or realization, of the booleans will be made more explicit when we consider the semantics of types later.

Definition *gt_code* :=

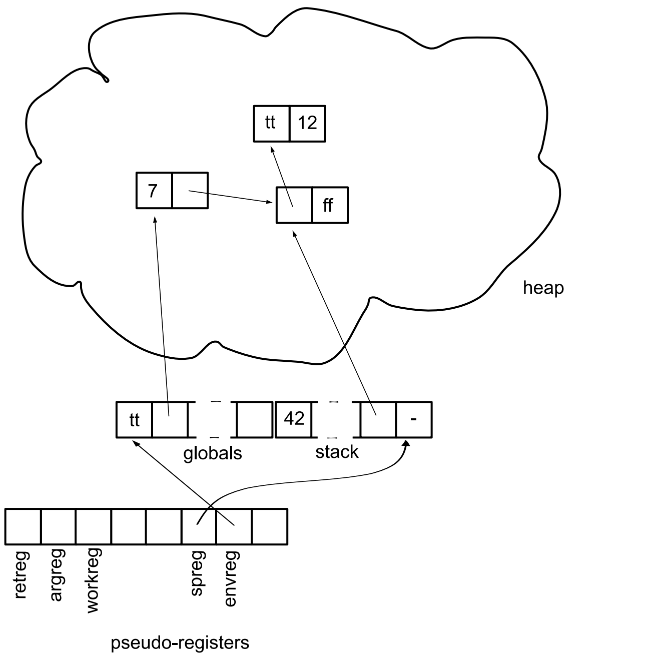

**Figure 2.** Memory Layout

  (*i_sub* (*d_imm spreg*) (*s_imm spreg*) (*s_cst* 2)) ::
  (*i_sub* (*d_ind spreg*) (*s_ind spreg*) (*s_indo* 1 *spreg*)) ::
  (*i_add* (*d_imm spreg*) (*s_imm spreg*) (*s_cst* 1)) ::
  *nil*.

The code for creating a pair has to allocate a fresh cons cell, pop two values off the stack and write them into the fields of the new cell and finally push the address of the new cell back to the stack. It is parameterized by the starting address of the code fragment, *label*, and the entry point of the allocation routine, *alloc*.

Definition *pair_code label alloc* :=
  (*i_sub* (*d_imm spreg*) (*s_imm spreg*) (*s_cst* 2)) ::
  (*i_move* (*d_imm argreg*) (*s_cst* 2)) ::
  (*i_move* (*d_imm retreg*) (*s_cst* (4 + *label*))) ::

```
  (i_branch (s_cst alloc)) ::
  (i_move (d_ind retreg) (s_ind spreg)) ::
  (i_move (d_indo 1 retreg) (s_indo 1 spreg)) ::
  (i_move (d_ind spreg) (s_imm retreg)) ::
  (i_add (d_imm spreg) (s_imm spreg) (s_cst 1)) ::
  nil.
```

The code sequences for addition and for projections are omitted, but may be found in the technical report.

We now show the function for compiling an expression *e*, given a starting address for the generated code, *label* and the address of the allocation routine, *alloc*. *compile_exp* returns a list of instructions and the next free code address. The *compile_expression* function wraps the compilation of a complete expression, decrementing the stack pointer at the end.

```
Fixpoint compile_exp (env:EnvType) (a:ExpType) (e:Exp env a)
(label alloc:nat) {struct e} : list instruction × nat:=
  match e with
    | EInt _ n ⇒ (int_code n, int_code_size + label)
    | EBool _ b ⇒ (bool_code b, bool_code_size + label)
    | EId _ n _ _ ⇒ (id_code n, id_code_size + label)
    | EAdd _ e1 e2 ⇒
        let (code',label') := compile_exp e1 label alloc in
        let (code",label") := compile_exp e2 label' alloc in
          (code' ++ code" ++ add_code,
           add_code_size + label")
    | EGt _ e1 e2 ⇒
        let (code',label') := compile_exp e1 label alloc in
        let (code",label") := compile_exp e2 label' alloc in
        (code' ++ code" ++ gt_code, gt_code_size + label")
    | EPair _ _ _ e1 e2 ⇒
        let (code',label') := compile_exp e1 label alloc in
        let (code",label") := compile_exp e2 label' alloc in
          (code' ++ code" ++ pair_code label" alloc,
          pair_code_size + label")
    | EFst _ _ _ e' ⇒
        let (code',label') := compile_exp e' label alloc in
        (code' ++ fst_code, fst_code_size + label')
    | ESnd _ _ _ e' ⇒
        let (code',label') := compile_exp e' label alloc in
          (code' ++ snd_code, snd_code_size + label')
  end.
```

```
Definition compile_expression env a (e:Exp env a) label alloc :=
  let (code, label) := compile_exp e label alloc in
      (code ++ ((i_sub (d_imm spreg) (s_imm spreg) (s_cst 1)) ::
nil), S label).
```

The *compile* function compiles a command:

```
Fixpoint compile (env1 env2: EnvType) (c:Command env1 env2)
(label alloc:nat) {struct c} : list instruction × nat:=
  match c with
    | CAssign _ m _ e ⇒
        let (code',label') := compile_expression e label alloc
        in (code' ++ (i_move (d_indo m envreg) (s_ind spreg)
            :: nil), 1 + label')
    | CSeq _ _ _ c1 c2 ⇒
        let (code',label') := compile c1 label alloc in
        let (code",label") := compile c2 label' alloc in
          (code' ++ code", label")
    | CIf _ _ b c1 c2 ⇒
       let (code',label') := compile_expression b label alloc in
        let (code",label") := compile c1 (1 + label') alloc in
        let (code"',label"') := compile c2 (1 + label") alloc in
          (code' ++ (i_brz (s_ind spreg) (s_cst (1 + label")) :: nil)
++ code" ++ (i_branch (s_cst label"') :: nil) ++
            code"', label"')
    | CWhile _ b c1 ⇒
      let (code',label') := compile_expression b label alloc in
        let (code",label") := compile c1 (1 + label') alloc in
        (code' ++ (i_brz (s_ind spreg) (s_cst (S label")) :: nil) ++
code" ++ (i_branch (s_cst label) :: nil), 1 + label")
  end.
```

We remark that (even without mutable pairs) the compiler *does* build datatstructures with non-trivial sharing. For example, the program

$$X := (3, 4)\ ;\ Y := (X, X)$$

generates two `cons` cells, with both fields of the second (which is pointed to from the variable $Y$) pointing to the first (which is also pointed to from $X$).

## 5. Relational Assertions

The next subsection gives a slightly informal account of the idea of relational specifications, which is followed by the more detailed Coq version.

### 5.1 Overview of relations for specification

The central idea of our approach to specifications in general, and the interpretation of types in particular, is that they are about invariance, independence, or 'how much difference makes a difference'. With our representations, there is no way that a statement like 'location 74 holds a boolean' can be interpreted as a predicate on the contents of location 74: whatever value $v$ is stored there, it is always interpretable as either a natural number, a boolean or even a pointer. How the value is interpreted depends on how it will be used, and the difference between a piece of code that is typed assuming location 74 holds a natural and one that is typed assuming that it holds a boolean is that the latter *should only care about* whether the value is zero or not. In other words, the code can have two different observable behaviours: one in the case that $v$ is zero and the other one for *all* the non-zero values. But the notion of observable behaviour needs to be defined carefully. Consider what we might mean by saying a piece of code is supposed to be both entered and exited with a boolean in location 74. This specification is met by code that does nothing, or which doubles the value in 74 (both of which implement, or realize, the identity on booleans). After the exit point however, we certainly *can* place a piece of code that behaves differently according to whether or not the initial value $v$ was, say, $42$. Clearly, we have to restrict the notion of allowable observation to take types into account, which we do by saying that *assuming* that the code at the exit point has the same behaviour for all non-zero values in location 74, *then* the code at the entry point promises to have the same behaviour whatever non-zero value is in 74 when *it* is called. We make this a bit more precise as follows. Define $[\![\text{bool}]\!]$ to be the binary relation

$$[\![\text{bool}]\!] \overset{def}{=} \{(n, n') \mid (n = n' = 0) \vee (n > 0 \wedge n' > 0)\}$$

capturing when two natural numbers are equivalent when interpreted as boolean values. Now, if $r \subseteq \mathbb{N} \times \mathbb{N}$ and $x \in \mathbb{N}$, define a relation on *state*s

$$(x \mapsto r) \overset{def}{=} \{(s, s') \mid (s\,x,\ s'\,x) \in r\}$$

So, in particular, $(74 \mapsto [\![\text{bool}]\!])$ relates two states when they hold values in location 74 that are $[\![\text{bool}]\!]$-related.

We now define the 'perp' operator, $(\cdot)^\top$, taking a binary relation on *state*s to one on pairs of programs and code pointers. If $R \subseteq$ *state* $\times$ *state*, then $R^\top$ relates two such pairs just when they behave equivalently whenever they are started in states that are $R$-related. The notion of equivalent behaviour we use here is *equi-termination*, defined using the *terminates* predicate from Section 2:

$$R^\top \overset{def}{=} \{((p,l),(p',l')) \mid \forall (s,s') \in R, \\ \textit{terminates}\ p\ s\ l \Leftrightarrow \textit{terminates}\ p'\ s'\ l'\}$$

One can think of the elements of $R^\top$ as 'test contexts' for $R$. The statement that a program fragment `M` both expects and produces a boolean in location 74 can now be expressed as:

$$\begin{aligned}&\forall p\, p',\ \textit{program\_extends\_frag}\ p\ \mathtt{M}\\ &\quad\Rightarrow \textit{program\_extends\_frag}\ p'\ \mathtt{M}\\ &\quad\Rightarrow ((p,\mathtt{exit}),(p',\mathtt{exit})) \in (74 \mapsto [\![\text{bool}]\!])^\top\\ &\quad\Rightarrow ((p,\mathtt{entry}),(p',\mathtt{entry})) \in (74 \mapsto [\![\text{bool}]\!])^\top.\end{aligned}$$

To be able to reason locally and modularly about relations on *store*s, we also need some handle on which *part* of the store a given relation depends upon, which we formalize in terms of invariance under change. If $L \subseteq \mathbb{N}$ and $s_0$ and $s_1$ are states, then define $s_0 \sim_L s_1$ to mean $\forall x \in L,\ s_0\, x = s_1\, x$. (In Coq, we represent subsets by maps into *Prop* and define *equpto* : $(\textit{nat} \to \textit{Prop}) \to \textit{state} \to \textit{state} \to \textit{Prop}$ to mean $\sim$.) Now, though we shall refine this definition shortly, say that a pair of sets of locations $(L, L')$ supports a relation $R \subseteq \textit{state} \times \textit{state}$ when

$$\forall (s_0,s_0') \in R,\ \forall s_1\, s_1',\ (s_0 \sim_L s_1) \wedge (s_0' \sim_{L'} s_1') \Rightarrow (s_1,s_1') \in R$$

In other words, if one starts with two states in the relation then any modifications outside the support yield another pair of states in the relation. If $R_1$ is supported by $(L_1, L_1')$ and $R_2$ by $(L_2, L_2')$, then define a form of separating conjunction [24] by

$$R_1 \otimes R_2 \overset{def}{=} \begin{cases} R_1 \cap R_2 & \text{if } L_1 \cap L_2 = \emptyset \text{ and } L_1' \cap L_2' = \emptyset\\ \emptyset & \text{otherwise}\end{cases}$$

So two states are in $R_1 \otimes R_2$ when they are in both $R_1$ and $R_2$ *and* the supports are disjoint. $R_1 \otimes R_2$ is supported by $(L_1 \cup L_2, L_1' \cup L_2')$. The separating conjunction allows concise specifications, as it abbreviates many 'absence of aliasing' conditions that would be explicit in a first-order assertion language. It proves even more useful when we reason about modules: the private invariants of modules will be captured by existentially quantifying over supported relations about which clients know nothing *except* that their support is disjoint from that of the client's own store.

Unfortunately, the above notion of support is slightly too weak. Consider a relation $(\textit{List}\, 74)$ expressing that two states have equal linked lists of integers starting from location 74. Assuming the usual representation, this will relate $s$ and $s'$ when *either* $s\,74$ and $s'\,74$ are both zero, *or* they are both non-zero, $s\,(s\,74) = s'\,(s'\,74)$, and (inductively) there are equal linked lists starting at $(s\,74)+1$ and $(s'\,74)+1$. Thus the sets of locations that get looked at depend on the contents of those stores. So we have to replace sets of locations $L \subseteq \mathbb{N}$ with functions $A : \textit{state} \to \mathbb{P}(\mathbb{N})$. We restrict attention to *accessibility maps* [12], those $A$ for which

$$\forall s\, s',\ s \sim_{A(s)} s' \Rightarrow A\, s = A\, s'$$

Intuitively, this says that $A$ 'supports itself', and makes relation $\sim_A$, defined by $s \sim_A s' \Leftrightarrow s \sim_{A(s)} s'$ an equivalence relation. We will build our specifications out of state relations supported by pairs of accessibility maps, making much use of (a generalization of) the separating conjunction.

## 5.2 Relations for specifications, formally

In this section we present the formal definitions of the relations and operations on relations with which we will be working. The first extra complexity compared with the semi-formal account above is that relations on *state*s and naturals will both generally depend on a pair of programs, because they will involve sets of code pointers that have particular behaviours, which only makes sense relative to some program. The second bit of structure we shall need is an admissibility property, to justify recursive reasoning about program fragments and definitions of relations. We get this by working with relations that are the limits of sequences of $k$-indexed approximants, where the natural number $k$ represents a number of steps in the operational semantics [7, 3, 9]. Thus our notions of 'equivalent' are expressed as the limit of 'indistinguishable for up to $k$ steps' as $k$ goes to $\omega$. As more steps allow more distinctions to be made, it is natural to work with indexed relations that are antimonotonic in $k$.

Here is the program- and step-indexed definition of relations on natural numbers. A *Natrel* is a record containing two fields. The first, *NRrel*, is the carrier: the relation itself. The second, *NRcond*, is a *proof* that the relation is antimonotonic in the index $k$:

```
Record Natrel : Type :=
  mkNR {NRrel :> program → program → nat → nat → nat →
Prop ;
          NRcond : ∀ p p' j k x x', j < k → NRrel p p' k x x' →
NRrel p p' j x x'}.
```

The carrier of a *Natrel* is a (curried) relation on pairs of programs and triples of naturals. The first two arguments are the left and right programs, *p* and *p'*. The third argument is the step index, *k*:*nat*. The fourth and fifth arguments are the naturals on the left and the right, *x* and *x'*. There is an order and an equality on *Natrel*s:

```
Definition Natrelleq (R1 R2 : Natrel) :=
  ∀ p p' k n n', R1 p p' k n n' → R2 p p' k n n'.
```

```
Definition Natreleq Na1 Na2 := Natrelleq Na1 Na2 ∧ Natrelleq
Na2 Na1.
```

We can also lift non-indexed relations on naturals to *Natrel*s:

```
Definition Natrel_lift (R : nat→nat → Prop) : Natrel.
  intro R.
  refine ( mkNR (fun p p' k ⇒ R) _ ).
  tauto.
Defined.
```

The definition of *Natrel_lift* makes use of the interactive proof language of Coq: the *refine* tactic is used to define the carrier of the lifted relation, leaving a hole (the underscore) for the *NRcond* proof component that is needed to show that the monotonicity requirement is satisfied. Rather than being constructed explicitly, the proof is then filled in interactively, in this simple case just by calling the automatic tactic *tauto*. We will henceforth elide the proof components of applications of *refine*.

Here is the definition of the type *Accrel* of supported, indexed relations on *state*s. An *Accrel* is a record comprising the relation itself (*ARrel*), two accessibility maps (*ARacc* and *ARacc'*), a proof (*ARcond*) that the accessibility maps are accessibility maps and do support the relation, and a proof (*ARindexed*) that the relation is antimonotonic in the step index:

```
Record Accrel : Type := mkAR {
  ARrel:>program→program→nat→state→state→Prop;
  ARacc : state → state → nat → Prop;
  ARacc' : state → state → nat → Prop;
  ARcond : ∀ p p' k s0 s0' s1 s1',
    (ARrel p p' k s0 s0') → equpto (ARacc s0 s0') s0 s1
                          → equpto (ARacc' s0 s0') s0' s1'
    → (ARrel p p' k s1 s1') ∧
       (∀ n, ARacc s0 s0' n ↔ ARacc s1 s1' n) ∧
       (∀ n, ARacc' s0 s0' n ↔ ARacc' s1 s1' n);
  ARindexed : ∀ p p' j k s s',
    j < k → ARrel p p' k s s' → ARrel p p' j s s' }.
```

The carrier relates two programs (*p* on the left, *p'* on the right), a step index and two states (*s* on the left, *s'* on the right). *ARacc* is the accessibility map giving the locations that are relevant on the left (i.e. in state *s*), whilst *ARacc'* is associated with the state on the right. Note that these are actually dependent on two states, rather than one as in our earlier overview; this turns out to be technically smoother, though we won't really exploit the extra generality here. *ARcond* combines conditions on accessibility maps and on the relation. Ignoring the program and index dependence it reads as follows: if we start with two states *s0* and *s0'* in the relation, and *s1* and *s1'* are two other states, with *s1* equal to *s0* up to the left hand accessibility map (applied to the states we started with), and *s1'* equal to *s0'* up to the right hand accessibility map, then three things happen. Firstly, *s1* and *s1'* are also in the relation - this says that the accessibility maps do support the relation. Second, the left hand accessibility map yields the same set of locations when given *s0* and *s0'* as arguments as it does when given *s1* and *s1'* - this is the accessibility map condition. Finally, the same holds of the other accessibility map. Compared with our informal account we have tied the maps and the relations closer together by only requiring the accessibility map condition on states in the relation.

*Accrel*s also have an equality and a partial order, involving implication between the carrier relations *and* a containment the other way between the accessibility maps:

```
Definition Accrelleq (Ar1 Ar2 : Accrel) :=
 ∀ p p' k s s', Ar1 p p' k s s' →
   ( (Ar2 p p' k s s') ∧
      (∀ n, ARacc Ar2 s s' n → ARacc Ar1 s s' n ) ∧
      (∀ n, ARacc' Ar2 s s' n → ARacc' Ar1 s s' n )).
```

```
Definition Accreleq Ar1 Ar2 := Accrelleq Ar1 Ar2 ∧ Accrelleq Ar2
Ar1.
```

We define *Emptyrel* (*q* : *Prop*) to be the *Accrel* with empty supports and a constant relation determined by *q*; in particular, *Toprel* := *Emptyrel True* is the constant true relation. *RelConj* is the (ordinary) additive conjunction on *Accrel*s, which allows sharing, so does not require disjoint supports:

```
Definition nunion (a1 a2 : nat→Prop) n :=
   (a1 n) ∨ (a2 n).
Definition RelConj (Ar1 Ar2 : Accrel) : Accrel.
   intros. refine (mkAR
     (fun p p' k s s' ⇒ (Ar1 p p' k s s') ∧
                          (Ar2 p p' k s s'))
     (fun s s' ⇒ (nunion (ARacc Ar1 s s')
                          (ARacc Ar2 s s')))
     (fun s s' ⇒ (nunion (ARacc' Ar1 s s')
                          (ARacc' Ar2 s s'))) _ _). . . .
Defined.
```

*RelTensor* is the multiplicative, separating conjunction:

```
Definition ndisj (a1 a2 : nat→Prop) :=
   ∀ n, ˜(a1 n ∧ a2 n).
Definition RelTensor (Ar1 Ar2 : Accrel) : Accrel.
   intros. refine (mkAR
     (fun p p' k s s' ⇒
       (Ar1 p p' k s s') ∧ (Ar2 p p' k s s') ∧
       (ndisj (ARacc Ar1 s s') (ARacc Ar2 s s')) ∧
       (ndisj (ARacc' Ar1 s s') (ARacc' Ar2 s s')))
     (fun s s' ⇒ (nunion (ARacc Ar1ss') (ARacc Ar2ss')))
     (fun s s' ⇒ (nunion (ARacc' Ar1 s s') (ARacc' Ar2 s s'))) _ _).
. . .
Defined.
```

Both *RelConj* and *RelTensor* are associative and commutative with *Toprel* as unit (amongst other properties whose formal statements we elide). The *ptsto* relation is like the 'points to' predicate of separation logic. It relates two states *s* and *s'* just when the values stored in location *l* in *s* and in location *l'* in *s'* are related by the *Natrel*, *r*.

```
Definition ptsto (l l':nat) (r : Natrel) : Accrel.
   intros.
   refine (mkAR (fun p p' k s s' ⇒ r p p' k (s l) (s' l'))
                (fun s s' n ⇒ n=l)
                (fun s s' n ⇒ n=l') _ _). . . .
Defined.
```

```
Notation "[ m , n ] |=> r" := (ptsto m n r) (at level 80).
Notation "m |-> r" := (ptsto m m r) (at level 80).
```

The definition of the 'perp' operation is the place where we make careful use of the step-indexing.

```
Definition Perp (R:Accrel) : Natrel.
   intros. refine( mkNR
     (fun p p' k l l' ⇒ ∀ j s s', j < k → R p p' j s s' →
         (((kstepterm j p s l) → (terminates p' s' l')) ∧
            ((kstepterm j p' s' l') → (terminates p s l))))
              _). . . .
Defined.
```

Note the way in which the indices are used: two labels $l$, $l'$ are in *Perp R* at index $k$ just when for any strictly smaller $j$, and states related by $R$ at index $j$, if jumping to $l$ terminates within $j$ steps, then jumping to $l'$ terminates in *some* number of steps, and vice versa. The limit of *Perp R* as $k$ goes to $\omega$ can be seen to agree with the definition of $R^{\top}$ that we gave earlier. As one would expect, *Perp* is contravariant:

```
Lemma Accrelleq_Perp : ∀ R1 R2,
   Accrelleq R1 R2 → Natrelleq (Perp R2) (Perp R1).
```

We define '#' as Coq notation for *RelTensor*, and '!' as notation for *Perp*.

## 6. Specification of Allocation

We briefly recall the specification of a memory allocator module from our previous work [10]. There are three entry points: for *init*ialization, for *alloc*ation, and for *dealloc*ation. We concentrate on allocation here, as we will not be using the other routines.

After the allocator is initialized, the heap will, like Gaul, be divided into three parts: the pseudo-registers 0 to 9, the part belonging to the allocator, and the part belonging to the rest of the program. Ownership of blocks is transferred between the allocator and its clients by calls to *alloc* and *dealloc*. The allocator promises not to (observably) read or write the part belonging to the clients, whilst the clients promise not to read or write the part belonging to the allocator *and* not to care about either the location or the initial contents of the blocks they are given.

We capture this intent by saying that a module $\mathtt{M}_a$ with entry point *alloc* meets the specification of an allocator if there exists a supported relation *Ra* – the allocator's private invariant – such that for all programs $p$, $p'$ extending $\mathtt{M}_a$, for all $k$, for all *Rc* (client invariants) and for all $n$ (block sizes),

$$(R\_al\ Ra\ n\ Rc)\ p\ p'\ k\ alloc\ alloc,$$

where

```
Definition R_aret (n:nat) : Accrel.
   intro. refine (mkAR
     (fun p p' k s s' ⇒ s 0 > 9 ∧ s' 0 > 9)
     (fun s s' l ⇒ (l = 0) ∨ (l ≥ s 0 ∧ l < n + s 0))
     (fun s s' l ⇒ (l = 0) ∨ (l ≥ s' 0 ∧ l < n + s' 0)) _ _). . . .
Defined.
```

```
Definition R_al (Ra:Accrel) n (Rc:Accrel) :=
! ((0 |-> !(R_aret n # T_rel (1 to 4) (1 to 4) # Rc # Ra))
  # (1 |-> (Natrel_lift (fun l l' ⇒ l = n ∧ l' = n)))
  # T_rel (2 to 4) (2 to 4) # Rc # Ra).
```

This means two calls to *alloc* must behave the same if they are started in initial states $s$, $s'$ that are related by all of the following disjoint relations: First, *Ra*, so the allocator's invariant holds before the call. Second, *Rc*, so the client's invariant holds before the call. Third, *T_rel* (2 *to* 4) (2 *to* 4). This is the 'true' relation with support $\{2, 3, 4\}$ on both sides, so these locations are not looked at by the allocator. Fourth, location 1 holds the value $n$ in both $s$ and $s'$. Fifth, the contents of location 0 on the two sides are code pointers that promise to behave the same if *they* are started in states related by all of the following: (i) *Ra*, so the allocator invariant holds *after* the call. (ii) *Rc*, so the client invariant holds after the call. (iii) *T_rel* (1 *to* 4) (1 *to* 4), so these locations are not looked at by the return addresses, i.e. they may be modified by the allocator. (iv) *R_aret n*, which expresses that location 0 on each side points to a block of size $n$ that doesn't overlap the pseudo-registers.

In previous work, we described a very naive allocation module that satisfies this specification; we have since verified that a slightly less trivial implementation that uses a free list satisfies the same spec.

## 7. Formalizing and Verifying Type Soundness

This section presents the actual type soundness theorem for the compiler. We start with a useful (if unusual-looking) construction on *Accrel*s:

```
Definition pex (l l':nat) (h: nat → nat → Accrel) : Accrel.
  intros. refine (mkAR
    (fun p p' k (s s':state) ⇒ h (s l) (s' l') p p' k s s' ∧
                         ¬ (ARacc (h (s l) (s' l')) s s' l) ∧
                         ¬ (ARacc' (h (s l) (s' l')) s s' l'))
    (fun s s' n ⇒ (n = l) ∨ (ARacc (h (s l) (s' l')) s s' n))
    (fun s s' n ⇒ (n = l') ∨ (ARacc' (h (s l) (s' l')) s s' n)) _ _). ...
Defined.
```

```
Notation "'pexists' ( x , y ) @ ( l , m ) , g" :=
  (pex l m (fun x y ⇒ g)) (at level 200, ...).
```

The *pexists* operation captures a pattern of existential quantification over values in the store that is common in defining *Accrel*s:

$$\exists\, x\, x',\ ([l, l'] \models> [x, x'])\ \#\ g(x, x').$$

States $s$ and $s'$ are in this relation when locations $l$ and $l'$ in $s$, $s'$ respectively hold some values $x$, $x'$ *and* the relation $g(x, x')$ holds on some disjoint part of the store.[2]

Now we define the semantics of types as relations over values and stores of the low-level machine:

```
Fixpoint typerefsem (t : ExpType) (l l' : nat) {struct t} : Accrel :=
  match t with
    | TInt ⇒ Emptyrel (l = l')
    | TBool ⇒ Emptyrel ((l=0) ∧ (l'=0) ∨ (l≠0) ∧ (l'≠0))
    | TPair a b ⇒ pexists (va, va') @ (l, l'), pexists (vb, vb') @ (S
l, S l'), RelConj (typerefsem a va va') (typerefsem b vb vb')
  end.
```

Two states are related by *typerefsem t l l'* just when *l* and *l'* are equal as values of type *t* in those states. So: (i) Two values are equal as natural numbers just when they are equal, independent of what the states are. (ii) Similarly, two values are equal as booleans just when they are in ⟦bool⟧, again independent of the states. (iii) $l$ and $l'$ are equal as values of type *TPair a b* in states $s$ and $s'$ when the cons cells pointed to by $l$ in $s$ and by $l'$ in $s'$ have first components that are equal as values of type *a* and second components that are equal as values of type *b*. The use of the additive conjunction, *RelConj*, allows the storage used by the values pointed to in the first and second components to share with one another, but note that we have not allowed sharing with the cell itself (because of the separation built into the definition of *pexists*).

The support part of the *Accrel* returned by *typerefsem* follows pointers to capture what parts of the two heaps are looked at in judging relatedness; this is a function of both the actual values in the heap and the type at which we are comparing them.

Having defined the relational interpretation of each *ExpType*, we need to define the relational interpretation of an *EnvType*, capturing the notion of equality on the vector of globals, the evaluation stack and the heap (as was illustrated in Figure 2). This is built up by induction over the length of the environment (globals+stack), taking care to maintain the separation between individual environment entries and between the environment and the heap, whilst allowing sharing within the heap. To this end, we first define a function that builds an *Accrel* by folding *pexists* over the vectors of length $n$ (starting at locations $l$ and $l'$, respectively) in the two states, additively conjoining all the results of applying a function $f$ to the existentially quantified values stored in the corresponding elements of the vectors.

```
Fixpoint pexconj (n m l l' : nat) (f : nat → nat → nat → Accrel)
(r:Accrel) {struct n} : Accrel :=
  match n with
    | 0 ⇒ r
    | S n' ⇒ pex l l' (fun (x1 x2 : nat) ⇒ pexconj n' (S m) (S l) (S
l') f (RelConj (f x1 x2 m) r))
  end.
```

Using *pexconj* to fold *typerefsem*, we can define the relational interpretation of an environment type *env* of length *n*, starting at locations *base* and *base'*:

```
Definition typesem n (env:EnvType) base base' :=
  pexconj n 0 base base' (fun x1 x2 m ⇒ typerefsem (env m) x1 x2)
Toprel.
```

Figure 3 shows an example of two states related by

$$typesem\ 4\ env\ 20\ 22$$

where we assume *env* maps offsets to types as follows:

| | | |
|---|---|---|
| 0 | $\mapsto$ | *TBool* |
| 1 | $\mapsto$ | (*TInt**TBool*)**(*TInt**TBool*) |
| 2 | $\mapsto$ | *TInt* |
| 3 | $\mapsto$ | *TInt**TBool* |

Having defined relations accounting for the structure of the environment and heap, we now need to define the contracts for the pieces of compiled code that come from typed expressions and commands in the source language. These will involve *Perp*s, expressing that jumping to certain pairs of addresses will yield equitermination provided that the initial states are in a certain relation, which will involve a *typesem* for the heap plus something about the pseudo-registers being suitably related.

Here is the formal definition of the prerelation for commands and expressions that expect to be entered with *envsize* global variables typed according to the *EnvType env* and an empty stack (which is allowed to grow up to *maxstack* $+ 1$ locations), assuming allocators related by *Ra* and starting addresses *envbase* and *envbase'* for the two environments. *Ro* is an arbitrary relation on the parts of memory which do *not* belong to either the allocator or the compiled code:

[2] The reason for this definition, due to Matthew Parkinson, is that supported relations do not admit general existential quantification. *pexists* is a form of quantification in which the witness is uniquely determined, fixing the support.

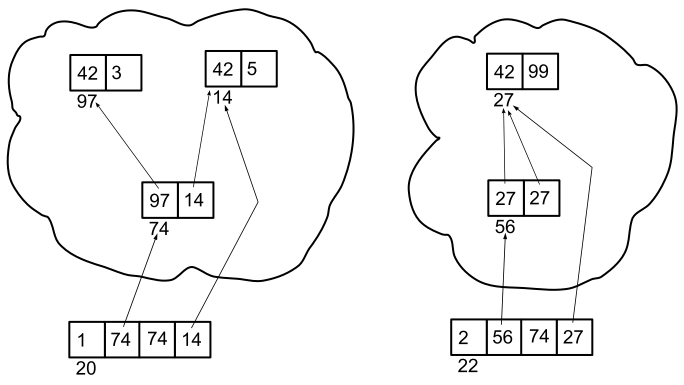

**Figure 3.** Typesem Example

Definition *R_comp* (*Ra Ro* : *Accrel*) *envsize env envbase envbase' maxstack* :=
  *let sp* := (*envsize* + *envbase*) *in*
  *let sp'* := (*envsize* + *envbase'*) *in*
  ! (*T_rel* (0 *to* 4) (0 *to* 4)
    # (*envreg* |-> (*Natrel_lift* (*fun l l'* ⇒ *l* = *envbase* ∧ *l'* = *envbase'*)))
    # (*spreg* |-> (*Natrel_lift* (*fun l l'* ⇒ *l* = *sp* ∧ *l'* = *sp'*)))
    # *typesem envsize env envbase envbase'*
    # *T_rel* (*sp to* (*maxstack*+*sp*)) (*sp' to* (*maxstack*+*sp'*))
    # *Ra* # *Ro*).

As explained in Section 5.1, however, the entry point of the code for a command or expression will only be in the *R_comp* corresponding to the pretype under the assumption that the code at its exit point satisfies a suitable relation for the posttype. For commands, which always expect to be entered with an empty stack, the assumption on the exit will just be another instance of *R_comp*. For expressions, however, we expect a value of a particular type to be left on the stack. That's expressed by the following variant of *R_comp* which adds the requirement that there be values related by the interpretation of type *t* on the stacks:

Definition *R_comp_exp_post* (*Ra Ro* : *Accrel*) *envsize* (*env* : *EnvType*) *t envbase envbase' maxstack* :=
  *let sp* := (*envsize* + *envbase*) *in*
  *let sp'* := (*envsize* + *envbase'*) *in*
  ! (*T_rel* (0 *to* 4) (0 *to* 4)
    # (*envreg* |-> (*Natrel_lift* (*fun l l'* ⇒ *l* = *envbase* ∧ *l'* = *envbase'*)))
    # (*spreg* |-> (*Natrel_lift* (*fun l l'* ⇒ *l* = *S sp* ∧ *l'* = *S sp'*)))
    # *typesem* (*S envsize*) (*envupdate env envsize t*) *envbase envbase'*
    # *T_rel* ((*S sp*) *to* (*maxstack* + *sp*)) ((*S sp'*) *to* (*maxstack* + *sp'*))
    # *Ra* # *Ro*).

Note the way in which the first stack location is treated as if it were the $(envsize+1)$-th variable. At last, we can give the type soundness theorems for our compiler. There is one for expressions and one for commands. Here is the one for expressions:

Theorem *comp_expression_thm* :
  ∀ (*alloc alloc'* : *nat*) (*Ra Ro* : *Accrel*) (*p p'* : *program*)
      (*envbase envbase'*: *nat*) (*envsize*: *nat*)
      (*env*:*EnvType*) (*a*:*ExpType*) (*e*:*Exp env a*),

  ∀ (*h_env*:*env_ok_exp e envsize*)
      (*maxstack*:*nat*) (*h_stack*:*stack_ok_exp e maxstack*)
      (*k label label'*: *nat*)
      (*code*:*list instruction*) (*code'*:*list instruction*)
      (*hcode*: *code* = *fst* (*compile_exp e label alloc*))
      (*hcode'*: *code'* = *fst* (*compile_exp e label' alloc'*)),

  *program_extends_frag p* (*fragfromlist code label*)
  → *program_extends_frag p'* (*fragfromlist code' label'*)
  → (∀ *n Rc*, (*R_al Ra n Rc*) *p p' k alloc alloc'*)
  → (*R_comp_exp_post Ra Ro envsize env a envbase envbase' maxstack*) *p p' k* (*length code* + *label*) (*length code'* + *label'*)

  → (*R_comp Ra Ro envsize env envbase envbase' maxstack*) *p p'* (1 + *k*) *label label'*.

where the function *fragfromlist* generates a *program_frag* from the instruction list returned by *compile_exp*, placing the first instruction at the given *label* argument.

Here is the theorem for commands:

Theorem *comp_thm* :
  ∀ (*alloc alloc'* : *nat*) (*Ra Ro* : *Accrel*) (*p p'* : *program*)
      (*envbase envbase'*: *nat*) (*envsize*: *nat*)
      (*env1 env2*:*EnvType*) (*c*:*Command env1 env2*),

  ∀ (*h_env*:*env_ok c envsize*)
      (*maxstack*:*nat*) (*h_stack*:*stack_ok c maxstack*)
      (*label label'*: *nat*)
      (*code*:*list instruction*) (*code'*:*list instruction*)
      (*hcode* : *code* = *fst* (*compile c label alloc*))
      (*hcode'* : *code'* = *fst* (*compile c label' alloc'*)),

  *program_extends_frag p* (*fragfromlist code label*)
  → *program_extends_frag p'* (*fragfromlist code' label'*)
  → ∀ *k*, ((∀ *n Rc*, (*R_al Ra n Rc*) *p p' k alloc alloc'*)
  → (*R_comp Ra Ro envsize env2 envbase envbase' maxstack*) *p p' k* (*length code* + *label*) (*length code'* + *label'*)
  → (*R_comp Ra Ro envsize env1 envbase envbase' maxstack*) *p p'* (1 + *k*) *label label'*).

Let's look at the theorem for commands, *comp_thm*, first to see what it says. Ignoring the checks that *maxstack* is sufficiently large and that only variables numbered less than *envsize* are used, the essence is the following:

- For any *Command*, *c*, typeable with a pretype *env1* and a posttype *env2*,
- if we compile *c* twice, once starting at *label* and once starting at *label'*, linking the first with an allocator at *alloc* and the second with an allocator at *alloc'*,
- then if we put those bits of code into contexts such that *alloc* and *alloc'* are equivalent memory allocators (according to the

specification of allocation) and the exit points of the two bits of compiled code behave equivalently in all states related by the interpretation of the posttype *env2*

- then the entry points of the bits of compiled code behave equivalently in all states related by the interpretation of the pretype *env1*.

The above captures lots of information about what the behaviour of the code compiled from *c* can depend upon. For example, it is independent of where the compiled code is placed, where the allocator is, where the environment is stored, what addresses the allocator returns and what their original contents are, how booleans or pairs are represented in the initial state, and so on.

We have a strong (extensional) form of memory safety, showing that the compiled code doesn't *observably* read or write any locations that it shouldn't. The preservation of any *Ro*, for example, means that storage disjoint from both the allocator's private store and the while-program's heap neither affects the behaviour of code compiled from a command, because the poststates will be equivalent for *any* initial *Ro*, nor is affected by it, because any *Ro* (in particular extensions of singleton relations) will be preserved. Note that the notion of independence really is more liberal than a naive intensional one: code that messes with unowned memory locations but restores them before exit meets the specification, as does code that builds literally different, but equivalent according to the types, heap structures according to the contents of unowned memory. See [11] for more on how preservation of sets of relations can express not only complete independence, but also read-only and write-only effects on particular storage locations.

The theorem for expressions is similar to that for commands, except that the environments in the pre and post relations stay the same and the postrelation assumes that there is a value of type *a* on the stack.

The proofs of the above theorems are basically inductions over the source language, with each case being dealt with by forward Hoare-style reasoning, similar to that of our previous work on allocation. The indexing structure on relations is used just in the case for *CWhile*, which uses mathematical induction: we assume that the label at the entry of the loop satisfies the desired relation at index $k$, and then examine the loop body to show that the entry then satisfies the same relation to index $k+1$.

The total size of the Coq development is around 8500 lines, which includes the low-level machine, metatheory of supported relations, the language and compiler and the actual proofs. There is scope for significant simplification here though. We are still comparatively inexperienced Coq users and were developing much of the theory *in* the prover, rather than doing post-hoc formalization of a completed paper development, so there is a lot of 'junk DNA' in those 8500 lines. We use little automation so far, but the proof scripts for particular segments of assembly code are already about an order of magnitude shorter than in our earlier efforts, averaging around 20 (instead of 200) lines of proof for each assembly language instruction.

## 8. Discussion

We have presented a semantic interpretation of the types of a high-level language as relations over configurations of a low-level machine, and used that to formulate and prove type correctness of a compiler.

A crucial feature of our approach is that the semantics of a type is a relation on low-level stores that makes no further reference to the original source language type. One might have instead defined a 'represents' relation between high-level values and low-level stores; two low-level stores could then be said to be equivalent at a type if there exists a high-level value such that both stores represent that value. We do not take such a definition as primitive (though we used something like it in some of the intuitive explanations in Section 7) for a couple of reasons. Firstly, it does not fit with our 'foundational' goal of compiling different high-level type systems to a common low-level assertion language in such a way that we can justify cross-language linking and specify run-time systems. Secondly, for languages with features such as higher-order functions and references, the question of what equal means at high-level types is about as hard, and addressed using the same relational techniques, as what we do at the low-level. Rather than construct a naive denotational semantics for the high-level language, then refine (quotient) it with a state-based logical relation [12] and *then* construct a relation between the refined model and the low level, we just construct low-level relations directly. This should ultimately prove simpler and more useful, since encapsulation provided by language features (e.g. local references) will be treated in the same way as encapsulation used in implementing language features (e.g. environments of closures, memory management). But that will only be tested when we consider more complex source languages.

The treatment of termination in the current work could be improved. Our use of perping means that an always-divergent program fragment satisfies any pre-post relation pair, irrespective of its effects on the store. This is normal for interpretations of types in languages with recursion or looping, but evaluation of *expressions* in this particular source language actually always terminates. One could make the low-level semantics closer to fully abstract by using relational total correctness judgements. There is a related weakness in the specification of allocation, which also allows for non-termination. These differences do not substantially weaken our type soundness result, but restrict our ability to prove program transformations. Type soundness is about programs being in the diagonal part of the relational interpretations of their types, i.e. being related to themselves. But we also want to prove that *different* pieces of machine code are equivalent modulo the contract of a particular type. If the allocator is assumed to be able to diverge, and we make different calls to the allocator in the two programs, then such proofs don't go through. Our more recent work uses an allocator specification that does enforce totality, so that we can reason about equations on low-level code.

There is a much related work, of which we can only mention a fraction. Compiler correctness has been studied for at least four decades [20] with notable early formalizations in the Boyer-Moore prover [33]. More recent examples include Leroy's verified compiler for a C-like language [19]. Full compiler correctness is more ambitious than type safety, but these projects relate high-level to low-level without the explicit language-independent low-level contracts that we are formalizing here.

Reasoning directly about unstructured low-level code also has a long history, going right back to Floyd's original work. The idea of developing type systems for low-level programs, and preserving typing through compilation, is more recent [21] and has attracted much attention in the context of proof-carrying code [23], as well as in more traditional compiler certification. That low-level types might be given a semantic interpretation using more primitive logical assertions is the key idea of *foundational* proof-carrying code [5].

Modelling types by partial equivalence relations goes back a long way [28, 4, 18, 2, 17] and, amongst many other things, parametric logical relations have recently been used by many authors in reasoning about program equivalences in the presence of higher order functions and dynamically allocated store [25, 27, 12]. Relational program logics have been developed by several researchers [1, 8, 32].

The other great influence on this work is separation logic [24, 29], though we work with relations rather than predicates, and use explicit higher-order parameterization over frames in place of the more usual 'tight' interpretation. Recent work on separation logic typing with higher order frame rules [15] and extensions with quantification [14, 16] are technically very close to the present work, though working on paper and with slightly higher-level languages. Hoare type theory (HTT) is a related mixture of polymorphism, dependent type theory and separation-logic style reasoning about side effects [22].

Thanks to Josh Berdine and Andrew Kennedy for useful discussions and feedback on earlier drafts of this work.

## References

[1] M. Abadi, L. Cardelli, and P.-L. Curien. Formal parametric polymorphism. *Theoretical Computer Science*, 121, 1993.

[2] M. Abadi and G. D. Plotkin. A PER model of polymorphism and recursive types. In *Proc. 5th IEEE Symposium on Logic in Computer Science (LICS)*, 1990.

[3] A. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

[4] R. M. Amadio. Recursion over realizability structures. *Information and Computation*, 91(1), 1991.

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

[6] A. Appel and A. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proc. 27th ACM Symposium on Principles of Programming Languages (POPL)*, 2000.

[7] 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.

[8] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proc. 31st ACM Symposium on Principles of Programming Languages (POPL)*, 2004. Revised version available from `http://research.microsoft.com/~nick/publications.htm`.

[9] N. Benton. A typed, compositional logic for a stack-based abstract machine. In *Proc. 3rd Asian Symposium on Programming Languages and Systems (APLAS)*, volume 3780 of *Lecture Notes in Computer Science*, 2005.

[10] N. Benton. Abstracting allocation: The new new thing. In *Proc. Computer Science Logic (CSL)*, volume 4207 of *Lecture Notes in Computer Science*, 2006.

[11] N. Benton, A. Kennedy, M. Hofmann, and L. Beringer. Reading, writing and relations: Towards extensional semantics for effect analyses. In *Proc. 4th Asian Symposium on Programming Languages and Systems (APLAS)*, number 4279 in Lecture Notes in Computer Science, 2006.

[12] 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.

[13] N. Benton and U. Zarfaty. Formalizing and verifying semantic type soundness for a simple compiler (preliminary report). Technical Report MSR-TR-2007-31, Microsoft Research, March 2007.

[14] B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctines and higher-order separation logic. In *Proc. 14th European Symposium on Programming (ESOP)*, volume 3444 of *Lecture Notes in Computer Science*, 2005.

[15] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation logic typing and higher-order frame rules. In *Proc. 20th IEEE Symposium on Logic in Computer Science (LICS)*, 2005.

[16] L. Birkedal and H. Yang. Relational parametricity and separation logic. In *Proc. 10th Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, volume 4423 of *Lecture Notes in Computer Science*, 2007.

[17] L. Cardelli and G. Longo. A semantic basis for Quest. In *Proc. ACM Conference on LISP and Functional Programming (LFP)*, 1990.

[18] F. Cardone. Relational semantics for recursive types and bounded quantification. In *Proc. International Colloquium on Automata, Languages and Programming (ICALP)*, volume 372 of *Lecture Notes in Computer Science*, 1989.

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

[20] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Mathematical Aspect of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*. AMS, 1967.

[21] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3), 1999.

[22] A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable ADTs in Hoare type theory. In *Proc. 16th European Symposium on Programming (ESOP)*, volume 4421 of *Lecture Notes in Computer Science*, 2007.

[23] G. Necula. Proof-carrying code. In *Proc. 24th ACM Symposium on Principles of Programming Languages (POPL)*, 1997.

[24] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proc. 10th Annual Conference of the European Association for Computer Science Logic (CSL)*, volume 2142 of *Lecture Notes in Computer Science*, 2001.

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

[26] G. D. Plotkin. Lambda definability and logical relations. Technical report, Department of AI, University of Edinburgh, 1973.

[27] U. S. Reddy and H. Yang. Correctness of data representations involving heap data structures. *Science of Computer Programming*, 50(1–3), 2004.

[28] J. C. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing '83*, 1983.

[29] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th IEEE Symposium on Logic in Computer Science (LICS)*, 2002.

[30] 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)*, 2004.

[31] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994.

[32] H. Yang. Relational separation logic. *Theoretical Computer Science*, 2004. Submitted.

[33] W. D. Young. A mechanically verified code generator. *J. Autom. Reason.*, 5(4), 1989.