Frame Rules from Answer Types for Code Pointers

Hayo Thielecke

School of Computer Science, University of Birmingham
Birmingham B15 2TT, United Kingdom
www.cs.bham.ac.uk/~hxt

Abstract

We define a type system, which may also be considered as a simple Hoare logic, for a fragment of an assembly language that deals with code pointers and jumps. The typing is aimed at local reasoning in the sense that only the type of a code pointer is needed, and there is no need to know the whole code itself. The main features of the type system are separation logic connectives for describing the heap, and polymorphic answer types of continuations for keeping track of jumps. Specifically, we address an interaction between separation and answer types: frame rules for local reasoning in the presence of jumps are recovered by instantiating the answer type. However, the instantiation of answer types is not sound for all types. To guarantee soundness, we restrict instantiation to closed types, where the notion of closedness arises from biorthogonality (in a sense inspired by Krivine and Pitts). A machine state is orthogonal to a disjoint heap if their combination does not lead to a fault. Closed types are sets of machine states that are orthogonal to a set of heaps. We use closed types as well-behaved answer types.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages, Theory, Verification

Keywords Code pointers, typed assembly language, Hoare logic, continuations, polymorphism

1. Introduction

Low-level programming languages, such as intermediate or assembly languages, are challenging to reason about, since the abstractions that one can rely on in high-level languages are not present: storage and pointers must be managed explicitly, and the control flow may be similarly unstructured. In recent years, there has been substantial progress in applying sophisticated type theories and logics to low-level languages, strongly motivated by verifying systems code and proof-carrying code [2, 19].

In particular, separation logic [9, 22] (see Reynolds's paper [27] for an overview) makes it possible to reason about heaps in a modular fashion: only the portion of the heap affected by the program fragment at hand needs to be considered, while frame rules can be used to infer that the remainder of the heap stays unchanged. The central idea in separation logic is the spatial conjunction $*$: a formula $P * Q$ is true in a given heap if the heap can be split into two disjoint subheaps, such that P holds in one and Q in the other. The heap-splitting semantics of $*$ then gives rise to frame rules: in a Hoare triple $\{P\}\ c\ \{Q\}$ for a command c, another formula R can be added with the separating conjunction:

$$\frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * R\}}$$

Any part of the heap that is not mentioned in the specification of the command c cannot be altered by it, so we can assume that R stays invariant (ignoring "modifies" clauses, which are not about the heap).

However, the very format of frame rules assumes purely functional control behaviour: c is assumed to have a single entry and exit point, to which we can then add R. If c could jump to some external label, the above rule would not be sound, unless $*R$ were also somehow added to the precondition of the label. (Recall that under the tight interpretation of separation logic it is generally not possible to jump to a label whose precondition is Q while passing along some extra heap, say $Q * R$.)

In assembly language, of course, all control is jumping. It appears, then, that in a language with arbitrary jumps we need to keep track of the control flow, perhaps using control flow analysis or a control effect system [10, 14]. The problem is in fact similar to effect masking in effect systems, in that we need to keep track of all possible exits from and entries into the code. However, it is not necessary to design a control effect system for assembly. Since everything is in continuation-passing style, we can use the answer types of continuations to keep track of jumps.

The notion of answer type of continuations may sound surprising to some readers in the light of the well-known classical typing [8] of control operators like `call/cc`. The type of `call/cc` is Peirce's law

$$((A \to \bot) \to A) \to A$$

where $\bot$ is some empty type that may logically be read as falsity. However, these classical typings are a feature of control operators in direct style. If everything is converted to continuation passing style, there are non-trivial answer types for continuations. The continuations never return to their point of call, but the answer type is not about returning to the point of call. Rather, it refers to the global answer of the whole program, or more operationally, its behaviour. A type like $A \multimap B$, for instance, can be read as stating that the code is requiring some new heap satisfying A, and will then behave according to B. The type B itself may again require more heap, or express a precondition on existing heap. If the last instruction in the code is a jump to some label f, then the type of f determines what the behaviour is going to be. In other words, the operation of jumping is polymorphic in the answer type.

The general idea of answer type polymorphism [29, 30] in the setting of the λ-calculus is as follows. An expression in continuation-passing style expects a continuation, which it may

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL'06 January 11–13, 2006, Charleston, South Carolina, USA.
Copyright © 2006 ACM 1-59593-027-2/06/0001...$5.00.

then apply to a value. If the expression has no control effects, it always applies the continuation; hence its type is of the form

$$\forall\alpha.(A \to \alpha) \to \alpha$$

where α is not free in A. The expression could also have a less rigid control behaviour. It may for example expect two continuations [28], exactly one of which must be called (say for successful and abnormal exit). This situation may be expressed with a typing like the following:

$$\forall\alpha.(A \to \alpha) \to (A' \to \alpha) \to \alpha$$

again assuming that α is not free in A or A'.

By instantiating the answer type, additional levels of state-passing can be added [29]. Let S be a type of stores, and instantiate α to $S \to \alpha'$. Then we have

$$\forall\alpha'.(A \to S \to \alpha') \to (A' \to S \to \alpha') \to S \to \alpha'$$

Note that the instantiation is done consistently for the two possible exits. Thus answer type polymorphism in continuation-passing style gives us a kind of semantic control flow analysis. It immediately makes powerful reasoning principles like parametricity [26] available. Specifically, equivalences have been shown for continuation passing [29, 30] using "Theorems for free" techniques [32].

The resemblance of answer type instantiation to frame rules may become a little more perspicuous with some uncurrying:

$$\forall\alpha'.(((A \times S) \to \alpha') \times ((A' \times S) \to \alpha') \times S) \to \alpha'$$

To be sure, the analogy to continuation passing in the λ-calculus is a little naive, because in a language with assignment not everything is as well-behaved as in the λ-calculus. In fact, if we read the $\times$ as analogous to logical conjunction, there is the evident problem that frame laws with (additive) conjunction rather than separating conjunction are unsound. That is the main problem that will have to be addressed: only some substitutions for the answer type can be allowed. These types correspond to frame rules with separating conjunction, but seen from the inside, so to speak, due to continuation-passing style.

Such types can be characterized as being orthogonal to a set of heaps, where a machine state is orthogonal to a heap if their combination does not lead to a run-time error. The general notion of orthogonality between a term and its context is due to Krivine, Pitts and others [11, 15, 23, 31], and is variously called biorthogonality, $\perp\perp$-closure, or $\top\top$-closure. In a sense, we generalize orthogonality from stacks to heaps. The connection to framing is that we define orthogonality using separating conjunction in such a way that only the interaction with disjoint heaps can be observed.

In brief, the contributions of this paper include the following:

- We give Hoare-style typing for code pointers with strong update;
- frame rules are derived from answer type polymorphism in continuation-passing style;
- the technique of biorthogonality (also called $\top\top$-closure) is adapted to a language with heaps.

1.1 Outline

The paper is organized as follows. We consider a small fragment of an assembly language; to clarify its intended meaning, we give an operational semantics in Section 2, before defining the type system in Section 3. We discuss closed types (in the sense of biorthogonality) in Section 4, as a preparation for giving a realizability semantics based on this idea on top of the operational semantics in Section 5. The main idea of this semantics is to recover frame rules, addressed in Section 6. We then discuss a limitation of the present paper, the absence of dynamic recursion through the heap (Section 7), and conclude with a discussion of related and possible future work in Section 8.

2. The language and operational semantics

We consider a very idealized fragment of an assembly language, with a straightforward operational semantics, which is fairly standard. Despite its simplicity, it is sufficient for our purpose here, since it contains operations for strong update of code pointers.

Code blocks (basic blocks) are sequences of instructions ending in a jump. They are defined by the following grammar:

$$\begin{array}{rcl} c & ::= & \texttt{jmp}\ f \\ & \mid & \texttt{jmp}\ [p] \\ & \mid & \texttt{movc}\ f\ p; c \\ & \mid & \texttt{movh}\ p\ q; c \end{array}$$

These instructions are the minimum we need for moving code pointers into the heap, updating them and jumping to them. The instruction `jmp` f jumps to a fixed label f in the code segment, while `jmp` $[p]$ jumps to a code pointer p in the heap (the brackets are intended to indicate indirect addressing). A code address f from the code segment can be stored in the heap using `movc` f p, while the instruction `movh` p q moves the contents of heap cells. We use c to range over basic blocks, f over immutable code addresses, and $p, q, \ldots$ over pointers.

A function mapping x to y is written as $x \mapsto y$, and we write $\mathrm{dom}(g)$ for the domain of definition of the function g. Both heaps and code segments will be modelled as such finite functions.

A program consists of a currently executing basic block and a finite mapping from addresses f_i to basic blocks c_i, which we write as

$$\{f_1 \mapsto c_1, \ldots, f_n \mapsto c_n\}$$

Strictly speaking, one could distinguish between the program in which the f_i are labels for the c_i and the mapping in memory at runtime; we conflate them for simplicity.

A machine state or configuration $\langle c \mid h \mid s\rangle$ consists of a current code block c, a mutable data heap h (which may include code pointers), and an immutable code segment s. (The executing code block is in reality the code pointed to by the instruction pointer, but writing it separately makes the operational semantics more readable.)

More precisely, such machine states with a currently executing code block will be called *active* states. In addition, we will also need a notion of *passive* state of the form $\langle h \mid s\rangle$ for a heap h and a code segment s.

States can be combined to form larger ones, provided that their code segments agree on their intersection and their heaps are disjoint (as required by the * operation on heaps). Two passive states together form another passive state, while an active and a passive state together form an active one. In the latter case, control starts off in the part coming from the active state, but could later pass to the formerly passive one.

The operational semantics is defined as a small-step transition relation $\leadsto$ between active machine states, given by the rules in Figure 1. Update of the heap cell p in h by some value x is written as $h[p \mapsto x]$.

There are two special code addresses exit and error, which are never in the domain of a code segment. There is no code associated with them, and the semantics gets stuck if a jump to either of them is attempted. Intuitively, one could think of these addresses as belonging to the surrounding operating system; jumping to them leaves the user code space and lets one observe termination, just as reduction to a value does in the context of a functional language. The operational semantics also gets stuck if a memory access outside the current heap or code segment is attempted.

$$\begin{array}{rcll}\langle \texttt{jmp}\ f \mid h \mid s\rangle & \leadsto & \langle s(f) \mid h \mid s\rangle & \text{where } f \notin \{\mathsf{exit}, \mathsf{error}\}\\ \langle \texttt{jmp}\ [p] \mid h \mid s\rangle & \leadsto & \langle s(h(p)) \mid h \mid s\rangle & \\ \langle \texttt{movc}\ f\ p; c \mid h \mid s\rangle & \leadsto & \langle c \mid h[p \mapsto f] \mid s\rangle & \\ \langle \texttt{movh}\ p\ q; c \mid h \mid s\rangle & \leadsto & \langle c \mid h[q \mapsto h(p)] \mid s\rangle & \end{array}$$

Figure 1. Operational semantics

It should be straightforward to add more details to this semantics in the form of registers and a richer instruction set for data, building on assembly languages from the literature [20]. (Since registers cannot be aliased, they are more innocuous than the heap from the point of view of separation.) However, the typing problems of code pointers that we are concerned with manifest themselves already in the semantics of this small fragment.

As an example of the operational semantics, consider the following transition sequence, where looping arises from a code pointer update:

$$\begin{array}{rl} & \langle \texttt{movc}\ f\ p; \texttt{jmp}\ f \mid p \mapsto g \mid f \mapsto \texttt{jmp}\ [p]\rangle \\ \leadsto & \langle \texttt{jmp}\ f \mid p \mapsto f \mid f \mapsto \texttt{jmp}\ [p]\rangle \\ \leadsto & \langle \texttt{jmp}\ [p] \mid p \mapsto f \mid f \mapsto \texttt{jmp}\ [p]\rangle \\ \leadsto & \langle \texttt{jmp}\ [p] \mid p \mapsto f \mid f \mapsto \texttt{jmp}\ [p]\rangle \\ \leadsto & \cdots \end{array}$$

3. Type system

The type system that we will use for the code pointer fragment is based on separation logic and bunched typing [21, 24]. It uses Hoare-style typing, in which assignment changes the types; alternatively, it may be seen as a fragment of Hoare logic with only simple type-like assertions. For simplicity, only a subset without recursion is considered, even though the operational semantics allows recursion, including recursion through the heap.

As with the operational semantics, the type system is only intended as a fragment to be integrated into a richer type theory. Its purpose is to provide a foundational interface with continuation-passing semantics. Derivations contain many $-\!\!*$ and $\forall$ introductions and eliminations that are operationally a "no-op", but more convenient idioms arise as derivable rules.

The main design decisions concern code pointers. The aim here is to reason locally only about the type (or specification) of a code pointer in the heap, without having to know what the actual code pointed to is.

We assume that the code segment is immutable, and hence not a resource in the same way that the data heap is. For the data heap, we specify *exactly* what the heap that we have access to contains. For the code segment, we only demand that it contain enough code of the right type, but it may contain more code that we do not (yet) know about. Of course, there may be situations when one wants to treat the code heap as a resource, for instance when dynamically loading code. But it would be cumbersome to have to treat the code in the same explicit way as the data heap all the time.

We distinguish between heap types A, behaviour types B and closed types C; see Figure 2 for their grammars.

Heap types use the standard separation logic connectives. The only novelty are points-to assertions of the form $p \mapsto B$, whose intended meaning is that the heap cell p points to code with type B. This does not mean that the machine instructions are in the heap; rather, p points to some heap cell containing an address in the code segment. B types specify the behaviour of code in terms of the arrow types of BI. They also include type variables α, which are used for answer type polymorphism. Other forms of polymorphism, in particular location polymorphism, would be natural features of a richer language, but are beyond the scope of the present paper.

Types of the form C, where the use of $\to$ is disallowed, form a subset of well-behaved behaviour types (the sense of well-behaved will be defined as $\perp\!\perp$-closedness in Section 4).

Definition 3.1 The rules of the type system are in Figure 3.

Basic blocks are typed with judgements of the form

$$\Gamma \mid A \vdash c : B$$

The intended meaning is that the basic block c may rely on code typed according to Γ, and a heap typed with A in order to behave as typed by B.

The rules for introduction and elimination of both arrow types ($\to$ and $-\!\!*$) are silent in that there is no syntax for λ-abstraction. They could also be written as two-way rules:

$$\frac{\Gamma \mid A' * A \vdash c : B}{\Gamma \mid A' \vdash c : A -\!\!* B} \qquad \frac{\Gamma \mid A' \wedge A \vdash c : B}{\Gamma \mid A' \vdash c : A \to B}$$

Since there are no λ-abstractions that bind any variables, arrow types are quite different from what may be familiar from functional languages. They do not introduce a function, but express the preconditions of a jump. An arrow type also does not delay execution, as it would in a call-by-value language.

Since we assume the code segment to be immutable, its typing is reminiscent of declarations. Code segments are typed using contexts of the form $f_1 \rhd B_1, \ldots, f_n \rhd B_n$. This typing ascribes the type B_i to the constant code pointer f_i. The code pointed to does not own any heap ($\texttt{emp}$), since we do not have closures. If the code wants to make any assumptions about the heap, it has to use arrow types.

Informally, one could read a type for a code pointer in the heap as an abbreviation for an existential

$$(p \mapsto B) \equiv \exists f.(p \mapsto f) \wedge (f \rhd B)$$

where the $(f \rhd B)$ is interpreted solely in the code segment, so that there is no conflict between the two pointers. The existential for the code address is left implicit in indirect addressing.

The typing rules contain the usual BI rules for weakening and contracting in a context $A(-)$, e.g., a heap $A * (A_1 \wedge A_1)$ can be contracted to $A * A_1$.

We also impose the following structural congruence, written as $\equiv$ and used by the rule ($\equiv$) on types:

- $\texttt{emp}$ and $*$ form a commutative monoid
- $\texttt{true}$ and $\wedge$ form a commutative monoid
- $((A * A') -\!\!* B) \equiv A -\!\!* (A' -\!\!* B)$
- $((A \wedge A') \to B) \equiv A \to (A' \to B)$

The interpretation of the separation logic connectives on the data heap is standard. Existing work, for instance dealing with

$$
\begin{array}{lcll}
A & ::= & p \mapsto B \mid A * A \mid A \wedge A \mid \texttt{emp} \mid \texttt{true} & \text{(heap types)} \\
B & ::= & A \multimap B \mid A \to B \mid \forall \alpha.B \mid \alpha & \text{(behaviour types)} \\
C & ::= & \alpha \mid A \multimap C \mid \forall \alpha.C & \text{(closed types)} \\
\\
\Gamma & ::= & - \mid \Gamma, f \triangleright B & \text{(code segments)}
\end{array}
$$

Figure 2. Syntax of types

Code blocks $\boxed{\Gamma \mid A \vdash c : B}$

$$\frac{}{\Gamma, f \triangleright B, \Gamma' \mid \texttt{emp} \vdash \texttt{jmp}\ f : B}\ (\textsc{Jmp})$$

$$\frac{}{\Gamma \mid p \mapsto \forall \alpha.((p \mapsto \alpha) \multimap C) \vdash \texttt{jmp}\ [p] : C}\ (\textsc{IndJmp}) \quad \text{where } \alpha \text{ is not free in } C$$

$$\frac{\Gamma \mid p \mapsto B * q \mapsto B \vdash c : C}{\Gamma \mid p \mapsto B * q \mapsto B' \vdash \texttt{movh}\ p\ q; c : C}\ (\textsc{MovHeap})$$

$$\frac{\Gamma, f \triangleright B, \Gamma' \mid p \mapsto B \vdash c : C}{\Gamma, f \triangleright B, \Gamma' \mid p \mapsto B' \vdash \texttt{movc}\ f\ p; c : C}\ (\textsc{MovCode})$$

$$\frac{\Gamma \mid A' * A \vdash c : B}{\Gamma \mid A' \vdash c : A \multimap B}\ ({\multimap}\text{I}) \qquad \frac{\Gamma \mid A' \vdash c : A \multimap B}{\Gamma \mid A' * A \vdash c : B}\ ({\multimap}\text{E})$$

$$\frac{\Gamma \mid A' \wedge A \vdash c : B}{\Gamma \mid A' \vdash c : A \to B}\ ({\to}\text{I}) \qquad \frac{\Gamma \mid A' \vdash c : A \to B}{\Gamma \mid A' \wedge A \vdash c : B}\ ({\to}\text{E})$$

$$\frac{\Gamma \mid A(A_1) \vdash c : B}{\Gamma \mid A(A_1 \wedge A_2) \vdash c : B}\ (\textsc{Weaken}) \qquad \frac{\Gamma \mid A(A_1 \wedge A_1) \vdash c : B}{\Gamma \mid A(A_1) \vdash c : B}\ (\textsc{Contr})$$

$$\frac{\Gamma \mid A \vdash c : B}{\Gamma \mid A' \vdash c : B'}\ (\equiv) \qquad \text{where } A \equiv A' \text{ and } B \equiv B'$$

$$\frac{\Gamma \mid A \vdash c \vdash B}{\Gamma \mid A \vdash c : \forall \alpha.B}\ (\forall\text{I}) \ \text{ where } \alpha \text{ is not free in } \Gamma \text{ or } A \qquad \frac{\Gamma \mid A \vdash c : \forall \alpha.B}{\Gamma \mid A \vdash c : B[C/\alpha]}\ (\forall\text{E})$$

Code segments $\boxed{\vdash s : \Gamma}$

$$\frac{}{\vdash \emptyset : -}\ (\textsc{Empty}) \qquad \frac{\vdash s : \Gamma \qquad \Gamma \mid \texttt{emp} \vdash c : B}{\vdash s \cup \{f \mapsto c\} : \Gamma, f \triangleright B}\ (\textsc{AddCode}) \ \text{ where } f \notin \mathrm{dom}(s)$$

Figure 3. Typing rules for the assembly language fragment

data structures as well as allocation and deallocation of storage, should be easy to adapt [4, 17]. The main difference from the languages typically used in the separation logic literature is that in assembly language there are no stack variables and everything is in the heap. This does not cause any additional difficulties, apart from the notational inconvenience of having to refer to the heap all the time.

3.1 Return addresses and indirect jumps

The rule for an indirect jump

$$\frac{}{\Gamma \mid p \mapsto \forall\alpha.((p \mapsto \alpha) \multimap C) \vdash \texttt{jmp}\ [p] : C}$$

deserves some explanation, since it is a deliberate restriction to avoid a knotty problem. When we jump to some code pointer in the heap, the same pointer is still in the heap, and so it is passed to the code. This is a form of self-application through the heap, so to type it in full generality, we would need recursive types. Since a semantic account of recursive types is beyond the scope of this paper, we compromise by restricting the indirect jumping to an idiom that avoids the recursion: the code pointed to by p may use p, but it cannot make any assumptions about the type of the code pointed to, due to the quantification. This restriction breaks the potential recursion. The idiom is motivated by return addresses: if p is some default location in which return addresses are passed in the function calling convention, then the code pointed to by p needs access to p to store the next return address when it calls the next function. That means we cannot hide p from the code using some sort of frame rule. However, in this idiom, all that the code does is overwrite p, so it can assume p to point to any type. Other idioms are possible, but return addresses are a useful one for our purposes here, since once we can type return addresses, function calls arise as an idiom, and frame rules for functions can then be formulated in Section 6. We revisit the more general form of jumping in Section 7.

3.2 Frames and answer types

The typing rules are akin to the so-called small axioms of separation logic. In essence, such axioms only mention the heap that changes. Whatever is left unchanged by the operation can then be added by frame laws. In our setting, all framing takes place at the answer type. For instance, consider the rule for storing a code pointer in the heap:

$$\frac{\Gamma \mid p \mapsto B \vdash c : C}{\Gamma \mid p \mapsto B' \vdash \texttt{movc}\ f\ p; c : C}$$

where Γ is of the form $\Gamma = \Gamma_1, f \triangleright B, \Gamma_2$.

This rule states that the heap must consist exactly of the code pointer p to be overwritten. Specializing C to $A \multimap C$, we can use the rule for cases where there is additional heap A.

$$\cfrac{\cfrac{\cfrac{\Gamma \mid (p \mapsto B) * A \vdash c : C}{\Gamma \mid p \mapsto B \vdash c : A \multimap C}\ (\multimap\text{I})}{\Gamma \mid p \mapsto B' \vdash \texttt{movc}\ f\ p; c : A \multimap C}\ (\textsc{MovCode})}{\Gamma \mid (p \mapsto B') * A \vdash \texttt{movc}\ f\ p; c : C}\ (\multimap\text{E})$$

However, we have to be careful how we specialize C. Suppose we allowed any type here, including $A \to C$. Then we could pick $p \mapsto B$ for A and infer:

$$\cfrac{\cfrac{\cfrac{\cfrac{\Gamma \mid (p \mapsto B) \wedge (p \mapsto B') \vdash c : C}{\Gamma \mid p \mapsto B \vdash c : (p \mapsto B') \to C}\ (\to\text{I})}{\Gamma \mid p \mapsto B' \vdash \texttt{movc}\ f\ p; c : (p \mapsto B') \to C}\ (\textsc{MovCode})}{\Gamma \mid (p \mapsto B') \wedge (p \mapsto B') \vdash \texttt{movc}\ f\ p; c : C}\ (\to\text{E})}{\Gamma \mid (p \mapsto B') \vdash \texttt{movc}\ f\ p; c : C}\ (\textsc{Contr})$$

Note that using $\to$ instead of $\multimap$ allows us to contract the disjunction $\wedge$ in the last step. This inference states that by storing f into p we can satisfy the precondition of the code c even though it may be inconsistent. For instance, if our logic includes assertions like $p \mapsto 1$, we could have $(p \mapsto 1) \wedge (p \mapsto 2)$ that holds for no heap.

Specializing answer types with $\multimap$ is analogous to frame laws (using $*$) in separation logic; specializing the answer with $\to$ amounts to unsound rules using $\wedge$ in place of $*$. Although the format of the inference above may look unfamiliar due to continuation-passing style, it is comparable to the unsoundness of $\wedge$ rather than $*$ in a putative frame law for traditional (direct-style) Hoare logic; a simple example of which is the following spurious inference:

$$\cfrac{\cfrac{\{x \mapsto 2\}\ [x] := 3\ \{x \mapsto 3\}}{\{x \mapsto 2 \wedge x \mapsto 2\}\ [x] := 3\ \{x \mapsto 2 \wedge x \mapsto 3\}}\ ???}{\{x \mapsto 2\}\ [x] := 3\ \{\texttt{false}\}}\ \text{Contraction}$$

The choice of possible answer types is behind many of the design decisions in the type system in Figure 3. Answer types include type variables α, but the rule for $\forall$-elimination only allows these variables to be instantiated to closed types C. The closed types do not include $\to$-types, thus avoiding the framing problem outlined above. The next section supports this choice with a more semantic view.

4. Closed types

We need to abstract from code by considering only its interaction with all disjoint heaps. To do so, we use a notion of $\perp\perp$-closure of a type, which includes all machine states that cannot be distinguished by the interaction with *disjoint* heaps.

It is not as evident as in functional languages what termination should mean in the context of assembly language. Here we assume that successful termination consists of a jump to a special label called exit. A notion of termination relative to a context will be built into an orthogonality relation $-\perp-$ analogous to the ones used by Krivine, Pitts and others [11, 15, 23]. The contexts are built from disjoint heaps, using separating conjunction. Since a heap may also contain pointers to known or unknown code, the heaps will be paired with a code segment that may overlap the current code segment; we refer to such pairs as passive states.

Definition 4.1 For heaps h and h', we write $h \# h'$ iff $\mathrm{dom}(h) \cap \mathrm{dom}(h') = \emptyset$.

For code segments s and s', we write $s \sharp s'$ iff $s(f) = s'(f)$ for all $f \in \mathrm{dom}(s) \cap \mathrm{dom}(s')$.

Since code segments are assumed immutable, we do not demand that they be disjoint, only that they do not disagree on any code in their possible overlap.

Definition 4.2 For an active state $\langle c \mid h \mid s\rangle$, we write

$$\langle c \mid h \mid s\rangle \updownarrow$$

if the state terminates successfully or loops, that is, if there is:

- a reduction $\langle c \mid h \mid s\rangle \rightsquigarrow^* \langle \texttt{jmp}\ \mathsf{exit} \mid h' \mid s'\rangle$ for some h' and s', or
- an infinite reduction $\langle c \mid h \mid s\rangle \rightsquigarrow \cdots \rightsquigarrow \cdots$

For an active state $\langle c \mid h \mid s\rangle$ and a passive state $\langle h' \mid s'\rangle$, we write

$$\langle c \mid h \mid s\rangle \perp \langle h' \mid s'\rangle$$

if $h \# h'$ and $s \sharp s'$ implies $\langle c \mid h * h' \mid s \cup s'\rangle \updownarrow$.

Intuitively, $\langle c \mid h \mid s\rangle \perp \langle h' \mid s'\rangle$ means that the active state is happy with the additional heap and code in the passive state provided they are compatible. Their combination may terminate successfully, or it may loop, but it will not "go wrong" by getting stuck in the operational semantics. We will need the following lemma about $\updownarrow$:

Lemma 4.3 If $\langle c \mid h \mid s\rangle \updownarrow$ and $s \subseteq s'$, then also $\langle c \mid h \mid s'\rangle \updownarrow$.

We use $\mathcal{A}$, $\mathcal{A}_1$ and $\mathcal{A}_2$ for sets of passive states, and $\mathcal{B}$ for sets of active states (since that is how in Section 5, types of the form A, respectively B, will be interpreted).

Given the definition of orthogonality, we define the biorthogonal or $\perp\perp$-closure of a set of states (analogous to similar definitions in the literature [11, 15, 23, 31]).

Definition 4.4 Let $\mathcal{A}$ be a set of passive states of the form $\langle h \mid s\rangle$, and $\mathcal{B}$ be a set of active states of the form $\langle c \mid h \mid s\rangle$. We define the orthogonal $\mathcal{A}^\perp$ of $\mathcal{A}$ and the orthogonal $\mathcal{B}^\perp$ of $\mathcal{B}$ as follows:

$$\begin{array}{rcl}\mathcal{A}^\perp & = & \{\langle c \mid h \mid s\rangle \mid \langle c \mid h \mid s\rangle \perp \langle h' \mid s'\rangle \\ & & \quad \text{for all } \langle h' \mid s'\rangle \in \mathcal{A}\}\end{array}$$

$$\begin{array}{rcl}\mathcal{B}^\perp & = & \{\langle h' \mid s'\rangle \mid \langle c \mid h \mid s\rangle \perp \langle h' \mid s'\rangle \\ & & \quad \text{for all } \langle c \mid h \mid s\rangle \in \mathcal{B}\}\end{array}$$

The set $\mathcal{B}^{\perp\perp}$ is called the biorthogonal or $\perp\perp$-closure of $\mathcal{B}$. $\mathcal{B}$ is called closed if $\mathcal{B}^{\perp\perp} \subseteq \mathcal{B}$.

As an example, consider an active state that loops, like

$$\Omega = \langle \texttt{jmp}\ f \mid \emptyset \mid \{f \mapsto \texttt{jmp}\ f\}\rangle$$

Then for any set of active states, $\mathcal{B}$, it is easy to see that $\Omega \in \mathcal{B}^{\perp\perp}$, since it is orthogonal to any set of passive states. Since we use the $\perp\perp$-closed sets as types, that is as it should be, just as a divergent term can have any value.

On the other hand, to see that not everything is identified, consider

$$\langle \texttt{jmp}\ [p] \mid \emptyset \mid \emptyset\rangle \text{ and } \langle \texttt{jmp}\ [q] \mid \emptyset \mid \emptyset\rangle$$

Then $\langle \texttt{jmp}\ [p] \mid \emptyset \mid \emptyset\rangle \notin \{\langle \texttt{jmp}\ [q] \mid \emptyset \mid \emptyset\rangle\}^{\perp\perp}$, as witnessed by

$$\langle \{q \mapsto f\} \mid \{f \mapsto \texttt{jmp exit}\}\rangle$$

The definition of $(-)^{\perp\perp}$ automatically entails a number of set-theoretic properties irrespective of the details of $-\perp-$; for instance, $(-)^{\perp\perp}$ is a closure operator [31]. Moreover, it enjoys some algebraic laws with regard to the separation logic connectives $*$ and $-\!*$, which we define as operations on sets of states:

Definition 4.5 Let $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}$ be sets of passive states, and $\mathcal{B}$ a set of active states. We define:

$$\begin{array}{rcl}\mathcal{A}_1 * \mathcal{A}_2 & = & \{\langle h_1 * h_2 \mid s_1 \cup s_2\rangle \mid h_1 \# h_2, s_1 \sharp s_2, \\ & & \quad \text{and } \langle h_i \mid s_i\rangle \in \mathcal{A}_i\} \\ \mathcal{A} -\!* \mathcal{B} & = & \{\langle c \mid h \mid s\rangle \mid \langle h' \mid s'\rangle \in \mathcal{A}, h \# h', s \sharp s' \\ & & \quad \text{implies } \langle c \mid h * h' \mid s \cup s'\rangle \in \mathcal{B}\} \\ \texttt{emp} & = & \{\langle \emptyset \mid s\rangle \mid s \text{ is any code segment}\}\end{array}$$

On the heap part of the states, this is the standard reading of the connectives. The definition of $(-)^\perp$ (Definition 4.4 above) is analogous to the one of $-\!*$ in its use of the implication. As one would expect, $-\!*$ is right adjoint to $*$:

Lemma 4.6 Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be sets of passive states and $\mathcal{B}$ a set of active states. Then we have:

$$(\mathcal{A}_1 * \mathcal{A}_2) -\!* \mathcal{B} = \mathcal{A}_1 -\!* (\mathcal{A}_2 -\!* \mathcal{B})$$

The set of all active states that do not go wrong is the same as $\texttt{emp}^\perp$, so that we have this lemma:

Lemma 4.7 Let $\mathcal{A}$ be a set of passive states. The operations $(-)^\perp$ and $-\!*$ satisfy:

- $\texttt{emp}^\perp = \{\langle c \mid h \mid s\rangle \mid \langle c \mid h \mid s\rangle \updownarrow\}$
- $\mathcal{A}^\perp = \mathcal{A} -\!* (\texttt{emp}^\perp)$.

Proof Let $\langle c \mid h \mid s\rangle \in \texttt{emp}^\perp$. As $\langle \emptyset \mid \emptyset\rangle \in \texttt{emp}$, this implies $\langle c \mid h * \emptyset \mid s \cup \emptyset\rangle \updownarrow$. Conversely, assume $\langle c \mid h \mid s\rangle \updownarrow$, and let $\langle \emptyset \mid s'\rangle \in \texttt{emp}$ with $s \sharp s'$. Then by Lemma 4.3,

$$\langle c \mid h * \emptyset \mid s \cup s'\rangle \updownarrow$$

Given the connection between $\updownarrow$ and $\texttt{emp}^\perp$, the identity $\mathcal{A}^\perp = \mathcal{A} -\!* (\texttt{emp}^\perp)$ is immediate. $\Box$

The next lemma gives two rules linking the separating connectives $*$ and $-\!*$ with the orthogonal $(-)^\perp$. Intuitively, an active state that can be combined with two disjoint passive states with types $\mathcal{A}_1$ and $\mathcal{A}_2$ is the same as an active state that, when first given an $\mathcal{A}_1$ state, could then be combined with an $\mathcal{A}_2$ state. A passive state of type $\mathcal{A}$ together with one that can be combined with an active one of type $\mathcal{B}$ amounts to a passive state that can be combined with an active state of type $\mathcal{A} -\!* \mathcal{B}$.

Lemma 4.8 Let $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}$ be sets of passive states and $\mathcal{B}$ a set of active states. Then:

$$\begin{array}{rcl}(\mathcal{A}_1 * \mathcal{A}_2)^\perp & = & \mathcal{A}_1 -\!* \mathcal{A}_2^\perp \\ \mathcal{A} * \mathcal{B}^\perp & \subseteq & (\mathcal{A} -\!* \mathcal{B})^\perp\end{array}$$

Proof $(\mathcal{A}_1 * \mathcal{A}_2)^\perp = \mathcal{A}_1 -\!* \mathcal{A}_2^\perp$ follows from $\mathcal{A}^\perp = \mathcal{A} -\!* \texttt{emp}^\perp$ and the fact that $-\!*$ is right adjoint to $*$:

$$\begin{array}{rl} & (\mathcal{A}_1 * \mathcal{A}_2)^\perp \\ = & (\mathcal{A}_1 * \mathcal{A}_2) -\!* \texttt{emp}^\perp \\ = & \mathcal{A}_1 -\!* (\mathcal{A}_2 -\!* \texttt{emp}^\perp) \\ = & \mathcal{A}_1 -\!* (\mathcal{A}_2^\perp)\end{array}$$

To prove $\mathcal{A} * \mathcal{B}^\perp \subseteq (\mathcal{A} -\!* \mathcal{B})^\perp$, we use the previous equality and the fact that $(-)^{\perp\perp}$ is a closure operator (so $\mathcal{X} \subseteq \mathcal{X}^{\perp\perp}$ for any $\mathcal{X}$).

Since $\mathcal{B} \subseteq \mathcal{B}^{\perp\perp}$ and $\mathcal{A} -\!* (-)$ preserves inclusions,

$$\begin{array}{rl} & \mathcal{A} -\!* \mathcal{B} \\ \subseteq & \mathcal{A} -\!* (\mathcal{B}^{\perp\perp}) \\ = & (\mathcal{A} * \mathcal{B}^\perp)^\perp\end{array}$$

Hence, since $(-)^\perp$ reverses inclusions,

$$\begin{array}{rl} & \mathcal{A} * \mathcal{B}^\perp \\ \subseteq & (\mathcal{A} * \mathcal{B}^\perp)^{\perp\perp} \\ \subseteq & (\mathcal{A} -\!* \mathcal{B})^\perp\end{array}$$

as required. $\Box$

Lemma 4.9 $\perp\perp$-closed types are closed under $\mathcal{A} -\!* (-)$; that is, if $\mathcal{C}$ is closed, then so is $\mathcal{A} -\!* \mathcal{C}$.

Proof Assume $\mathcal{C}$ is closed, that is, $\mathcal{C}^{\perp\perp} \subseteq \mathcal{C}$. We have to show that this implies $(\mathcal{A} -\!* \mathcal{C})^{\perp\perp} \subseteq \mathcal{A} -\!* \mathcal{C}$. By Lemma 4.8, we have $(\mathcal{A} * \mathcal{C}^\perp) \subseteq (\mathcal{A} -\!* \mathcal{C})^\perp$. As $(-)^\perp$ reverses inclusions and $\mathcal{A} -\!* (-)$ preserves them, we infer from that:

$$(\mathcal{A} -\!* \mathcal{C})^{\perp\perp}$$

$$\begin{array}{rl} \subseteq & (\mathcal{A} * \mathcal{C}^{\perp})^{\perp} \\ = & \mathcal{A} \mathrel{-\!\!*} \mathcal{C}^{\perp\perp} \\ \subseteq & \mathcal{A} \mathrel{-\!\!*} \mathcal{C} \end{array}$$

and hence the desired inclusion. □

Lemma 4.9 is crucial for our approach to frame rules by way of instantiation of answer type variables by closed types. What we have established in this section is a well-behaved and sufficiently rich set of possible answer types, which the semantics in the next Section will build on.

5. Realizability semantics

Using the notion of $\perp\perp$-closed set from Section 4, we can now define a semantics for the type system from Figure 3 in Section 3. Each type is interpreted as a set of untyped realizers. More specifically, a type of the form A is interpreted as a set of passive states, and a behaviour type B as a set of active states (much as in Definition 4.5) . A context Γ is interpreted as a set of code segments. All interpretations are relative to a type environment η that maps type variables to sets of active states. We require the type environment to be closed, that is, for each α, $\eta(\alpha)$ is a $\perp\perp$-closed set of active states.

Definition 5.1 Let η be a closed type environment. We define a realizability semantics of behaviour types $[\![B]\!]\,\eta$, heap types $[\![A]\!]\,\eta$ and contexts $[\![\Gamma]\!]\,\eta$ by the rules given in Figure 4.

A subtle point in the semantics is the definition of $[\![A \mathrel{-\!\!*} B]\!]\,\eta$. As far as the heap is concerned, the definition follows the standard semantics of separation logic. However, the code segment is treated differently. When the code of type $[\![A \mathrel{-\!\!*} B]\!]\,\eta$ is supplied with a heap satisfying $[\![A]\!]\,\eta$, the code pointers in the new heap may point to the code segment, or to previously unknown code.

Quantification $\forall\alpha.B$ ranges only over closed types. The next lemma justifies calling types of the form C closed.

Lemma 5.2 For any type of the form

$$C ::= \alpha \mid A \mathrel{-\!\!*} C \mid \forall\alpha.C$$

and closed type environment η, $[\![C]\!]\,\eta$ is $\perp\perp$-closed.

Proof $[\![\alpha]\!]\,\eta$ is $\perp\perp$-closed by definition. The $\perp\perp$-closed sets are closed under $\mathcal{A} \mathrel{-\!\!*} (-)$ by Lemma 4.9, and they are closed under arbitrary intersections for set-theoretic reasons independently of the definition of $-\perp-$ □

Lemma 5.3 The adjoint isomorphisms are equalities:

$$\begin{array}{rcl} [\![(A_1 * A_2) \mathrel{-\!\!*} B]\!]\,\eta & = & [\![A_1 \mathrel{-\!\!*} (A_2 \mathrel{-\!\!*} B)]\!]\,\eta \\ [\![(A_1 \wedge A_2) \to B]\!]\,\eta & = & [\![A_1 \to (A_2 \to B)]\!]\,\eta \end{array}$$

The code segment is treated intuitionistically: moving from a code segment s to a larger one s' with $s \subseteq s'$ does not change the type, unlike the tight interpretation of data heaps.

Lemma 5.4 Let $s \subseteq s'$ be code segments. Then:

- $\langle c \mid h \mid s\rangle \in [\![B]\!]\,\eta$ implies $\langle c \mid h \mid s'\rangle \in [\![B]\!]\,\eta$
- $\langle h \mid s\rangle \in [\![A]\!]\,\eta$ implies $\langle h \mid s'\rangle \in [\![A]\!]\,\eta$
- $s \in [\![\Gamma]\!]\,\eta$ implies $s' \in [\![\Gamma]\!]\,\eta$

Lemma 5.5 If $s \in [\![\Gamma, f \rhd B, \Gamma']\!]\,\eta$, then $\langle s(f) \mid \emptyset \mid s\rangle \in [\![B]\!]\,\eta$.

Lemma 5.6 Let B be a behaviour type. If $\langle c \mid h \mid s\rangle \rightsquigarrow \langle c' \mid h \mid s\rangle$ and $\langle c' \mid h \mid s\rangle \in [\![B]\!]\,\eta$, then also $\langle c \mid h \mid s\rangle \in [\![B]\!]\,\eta$.

With the help of the preceding lemmas, we now prove soundness. The type system in Figure 3 is sound with respect to the realizability semantics in Figure 4. If we can infer a type for code in a judgement $\Gamma \mid A \vdash c : B$, then the code realizes B whenever it is placed in a machine state that is equipped with a code segment that realizes the code context Γ and a heap that realizes A.

Theorem 5.7 Let η be a closed type environment.

- If $\Gamma \mid A \vdash c : B$ is derivable, then $s \in [\![\Gamma]\!]\,\eta$ and $\langle h \mid s\rangle \in [\![A]\!]\,\eta$ implies $\langle c \mid h \mid s\rangle \in [\![B]\!]\,\eta$.
- If $\vdash s : \Gamma$ is derivable, then $s \in [\![\Gamma]\!]\,\eta$.

Proof (sketch) The proof proceeds by induction over the derivation of typing judgements. Only a few cases are given here, emphasising the role of closed types.

First, consider the typing rule for moving a label into a code pointer:

$$\frac{\Gamma, f \rhd B, \Gamma' \mid p \mapsto B \vdash c : C}{\Gamma, f \rhd B, \Gamma' \mid p \mapsto B' \vdash \texttt{movc}\ f\ p; c : C}\ (\textsc{MovCode})$$

Let $s \in [\![\Gamma, f \rhd B, \Gamma']\!]\,\eta$ and $\langle h \mid s\rangle \in [\![p \mapsto B']\!]\,\eta$. The latter implies that $h = \{p \mapsto f'\}$ for some f'. We need to show that

$$\langle \texttt{movc}\ f\ p; c \mid \{p \mapsto f'\} \mid s\rangle \in [\![C]\!]\,\eta$$

Since $[\![C]\!]\,\eta$ is $\perp\perp$-closed by Lemma 5.2, it is sufficient to show membership of $([\![C]\!]\,\eta)^{\perp\perp}$. So suppose we have some $\langle h_1 \mid s_1\rangle \in ([\![C]\!]\,\eta)^{\perp}$ with $\mathrm{dom}(h) \cap \mathrm{dom}(h_1) = \emptyset$ and $s(g) = s_1(g)$ for all $g \in \mathrm{dom}(s) \cap \mathrm{dom}(s_1)$. We need to show that

$$\langle \texttt{movc}\ f\ p; c \mid \{p \mapsto f'\} * h_1 \mid s \cup s_1\rangle \updownarrow$$

The first transition step of this state is:

$$\begin{array}{rl} & \langle \texttt{movc}\ f\ p; c \mid \{p \mapsto f'\} * h_1 \mid s \cup s_1\rangle \\ \rightsquigarrow & \langle c \mid \{p \mapsto f\} * h_1 \mid s \cup s_1\rangle \end{array}$$

Now $\langle \{p \mapsto f\} \mid s\rangle \in [\![p \mapsto B]\!]\,\eta$ due to $s \in [\![\Gamma, f \rhd B, \Gamma']\!]\,\eta$. So by the induction hypothesis, $\langle c \mid \{p \mapsto f\} \mid s\rangle \in [\![C]\!]\,\eta$. Since $\langle h_1 \mid s_1\rangle \in ([\![C]\!]\,\eta)^{\perp}$, that implies

$$\langle c \mid \{p \mapsto f\} * h_1 \mid s \cup s_1\rangle \updownarrow$$

so $\langle \texttt{movc}\ f\ p; c \mid \{p \mapsto f'\} * h_1 \mid s \cup s_1\rangle \updownarrow$ as well. As this holds for any $\langle h_1 \mid s_1\rangle \in ([\![C]\!]\,\eta)^{\perp}$, we have that

$$\langle \texttt{movc}\ f\ p; c \mid \{p \mapsto f'\} \mid s\rangle \in ([\![C]\!]\,\eta)^{\perp\perp} \subseteq [\![C]\!]\,\eta$$

as required.

Next, consider the rule for an indirect jump along a code pointer:

$$\frac{}{\Gamma \mid p \mapsto \forall\alpha.((p \mapsto \alpha) \mathrel{-\!\!*} C) \vdash \texttt{jmp}\ [p] : C}\ (\textsc{IndJmp})$$

We assume that α is not free in C. Let $s \in [\![\Gamma]\!]\,\eta$ and $\langle h \mid s\rangle \in [\![p \mapsto \forall\alpha.((p \mapsto \alpha) \mathrel{-\!\!*} C]\!]\,\eta$. Then we have $h = \{p \mapsto f\}$ for some f with $s(f) = c$ such that

$$\langle c \mid \emptyset \mid s\rangle \in [\![\forall\alpha.((p \mapsto \alpha) \mathrel{-\!\!*} C]\!]\,\eta$$

Let $\mathcal{C} = [\![\forall\alpha.((p \mapsto \alpha) \mathrel{-\!\!*} C)]\!]\,\eta$. Then

$$\begin{array}{rcl} \langle c \mid \emptyset \mid s\rangle & \in & [\![(p \mapsto \alpha) \mathrel{-\!\!*} C]\!]\,(\eta[\alpha \mapsto \mathcal{C}]) \\ & = & [\![p \mapsto \forall\alpha.((p \mapsto \alpha) \mathrel{-\!\!*} C)]\!]\,\eta \mathrel{-\!\!*} [\![C]\!]\,\eta \end{array}$$

Furthermore, the next transition step causes c to run:

$$\langle \texttt{jmp}\ [p] \mid \{p \mapsto f\} \mid s\rangle \rightsquigarrow \langle c \mid \{p \mapsto f\} \mid s\rangle$$

Since $\langle \{p \mapsto f\} \mid s\rangle \in [\![p \mapsto \forall\alpha.((p \mapsto \alpha) \mathrel{-\!\!*} C)]\!]\,\eta$ and trivially $s \sharp s$, we then have by the definition of $\mathrel{-\!\!*}$ that

$$\langle c \mid \{p \mapsto f\} \mid s\rangle \quad \in \quad [\![C]\!]\,\eta$$

$$\begin{array}{rcll}
\langle c \mid h \mid s\rangle & \in & \llbracket A \to B \rrbracket\,\eta & \text{iff} \quad \langle h \mid s'\rangle \in \llbracket A \rrbracket\,\eta \text{ and } s \subseteq s' \text{ implies } \langle c \mid h \mid s'\rangle \in \llbracket B \rrbracket\,\eta \\
\langle c \mid h \mid s\rangle & \in & \llbracket A \mathrel{-\!\!*} B \rrbracket\,\eta & \text{iff} \quad \langle h' \mid s'\rangle \in \llbracket A \rrbracket\,\eta \text{ with } h \mathrel{\#} h' \text{ and } s \sharp s' \\
 & & & \qquad \text{implies } \langle c \mid h * h' \mid s \cup s'\rangle \in \llbracket B \rrbracket\,\eta \\
\langle c \mid h \mid s\rangle & \in & \llbracket \alpha \rrbracket\,\eta & \text{iff} \quad \langle c \mid h \mid s\rangle \in \eta(\alpha) \\
\langle c \mid h \mid s\rangle & \in & \llbracket \forall\alpha.B \rrbracket\,\eta & \text{iff} \quad \text{for all } \perp\!\perp\text{-closed sets } \mathcal{C} \text{ of active states }, \langle c \mid h \mid s\rangle \in \llbracket B \rrbracket\,(\eta[\alpha \mapsto \mathcal{C}]) \\
\langle h \mid s\rangle & \in & \llbracket p \mapsto B \rrbracket\,\eta & \text{iff} \quad h = \{p \mapsto f\} \text{ and } s(f) = c \text{ with } \langle c \mid \emptyset \mid s\rangle \in \llbracket B \rrbracket\,\eta \\
\langle h \mid s\rangle & \in & \llbracket A_1 \wedge A_2 \rrbracket\,\eta & \text{iff} \quad \langle h \mid s\rangle \in \llbracket A_1 \rrbracket\,\eta, \langle h \mid s\rangle \in \llbracket A_2 \rrbracket\,\eta \\
\langle h \mid s\rangle & \in & \llbracket A_1 * A_2 \rrbracket\,\eta & \text{iff} \quad \langle h_1 \mid s_1\rangle \in \llbracket A_1 \rrbracket\,\eta, \langle h_2 \mid s_2\rangle \in \llbracket A_2 \rrbracket\,\eta \\
 & & & \qquad \text{where } h = h_1 * h_2, s_1 \cup s_2 = s \text{ with } h_1 \mathrel{\#} h_2 \text{ and } s_1 \sharp s_2 \\
\langle h \mid s\rangle & \in & \llbracket \mathtt{emp} \rrbracket\,\eta & \text{iff} \quad h = \emptyset \\
\langle h \mid s\rangle & \in & \llbracket \mathtt{true} \rrbracket\,\eta & \text{iff} \quad h \text{ is any heap} \\
s & \in & \llbracket \Gamma, f \rhd B \rrbracket\,\eta & \text{iff} \quad s \in \llbracket \Gamma \rrbracket\,\eta \text{ and } s(f) = c \text{ with } \langle c \mid \emptyset \mid s\rangle \in \llbracket B \rrbracket\,\eta \\
s & \in & \llbracket - \rrbracket\,\eta & \text{iff} \quad s \text{ is any code segment}
\end{array}$$

Figure 4. Realizability of types

Due to the transition

$$\langle \mathtt{jmp}\ [p] \mid \{p \mapsto f\} \mid s\rangle \leadsto \langle c \mid \{p \mapsto f\} \mid s\rangle$$

we have $\langle \mathtt{jmp}\ [p] \mid \{p \mapsto f\} \mid s\rangle \in \llbracket C \rrbracket\,\eta$ as well, and we are done with this case.

For a direct jump, we do not need to assume a closed answer type:

$$\frac{}{\Gamma, f \rhd B, \Gamma' \mid \mathtt{emp} \vdash \mathtt{jmp}\ f : B}\ (\textsc{Jmp})$$

Suppose $s \in \llbracket \Gamma, f \rhd B, \Gamma' \rrbracket\,\eta$ and $\langle h \mid s\rangle \in \llbracket \mathtt{emp} \rrbracket\,\eta$. Now $\langle h \mid s\rangle \in \llbracket \mathtt{emp} \rrbracket\,\eta$ implies $h = \emptyset$ and $s \in \llbracket \Gamma, f \rhd B, \Gamma' \rrbracket\,\eta$ implies $\langle s(f) \mid \emptyset \mid s\rangle \in \llbracket B \rrbracket\,\eta$. The next machine transition is:

$$\langle \mathtt{jmp}\ f \mid \emptyset \mid s\rangle \quad \leadsto \quad \langle s(f) \mid \emptyset \mid s\rangle$$

By Lemma 5.6, that implies $\langle \mathtt{jmp}\ f \mid h \mid s\rangle \in \llbracket B \rrbracket\,\eta$.

For building up code segments from code blocks, we need to show the soundness of (AddCode). Suppose $\vdash s : \Gamma$ and $\Gamma \mid \mathtt{emp} \vdash c : B$. By the induction hypothesis applied to $\vdash s : \Gamma$, we have $s \in \llbracket \Gamma \rrbracket\,\eta$ and so $\langle c \mid \emptyset \mid s\rangle \in \llbracket B \rrbracket\,\eta$. Let $s' = s \cup \{f \mapsto c\}$. Then by Lemma 5.4, we have also $s' \in \llbracket \Gamma \rrbracket\,\eta$ and $\langle c \mid \emptyset \mid s'\rangle \in \llbracket B \rrbracket\,\eta$, and therefore $s' \in \llbracket \Gamma, f \rhd B \rrbracket\,\eta$, as required.

□

6. Frame rules

Instantiation of the answer type to types of the form $A \mathrel{-\!\!*} C$ gives us frame rules whose soundness follows from the soundness of the type system, Theorem 5.7. In particular, we define function types as the evident continuation-passing type with a return address r:

$$\begin{array}{rl} & A_1 \Rightarrow_r A_2 \\ \equiv & \forall\alpha.(r \mapsto (\forall\alpha_r.(r \mapsto \alpha_r) \mathrel{-\!\!*} A_2 \mathrel{-\!\!*} \alpha)) \mathrel{-\!\!*} A_1 \mathrel{-\!\!*} \alpha \end{array}$$

The answer type polymorphism [29, 30] in this idiom means that the function must eventually invoke its return continuation or loop; it cannot jump to a different continuation instead.

We then have a derivable frame rule for the function call idiom:

$$\frac{}{\Gamma, f \rhd A_1 \Rightarrow_r A_2, \Gamma' \vdash \mathtt{jmp}\ f : (A_1 * A') \Rightarrow_r (A_2 * A')}$$

See Figure 5 for the derivation, which uses instantiation of the answer type variable and congruence for $\mathrel{-\!\!*}$ types.

Frame laws for continuation passing with quantified answer types are not restricted to purely functional idioms. Figure 6 shows how to frame in $A * -$ for an indirect jump of the form $\mathtt{jmp}\ [p]$. Syntactically, the derived rule looks different from the usual Hoare logic one, since there are two $- * A$ on the left of the $\vdash$; they are however in a covariant and a contravariant position.

Using the function type idiom $\Rightarrow_r$ with the return address passed in r, the identity function $(\lambda x.x)$ can be compiled into the code $\mathtt{jmp}\ [r]$. We type it as:

$$- \mid \mathtt{emp} \vdash \mathtt{jmp}\ [r] : A \Rightarrow_r A$$

We then have the typing

$$\vdash \{f \mapsto \mathtt{jmp}\ [r]\} : f \rhd A \Rightarrow_r A$$

for the code segment holding the identity function at address f.

To write more involved examples, it would be useful to extend the language and typing fragment with more features that make it less cumbersome to pass around pointers. Even the typing for $\mathtt{jmp}\ [p]$ used so far is a restriction to a special case, and we return to this point in the next section.

7. Recursion through the heap

The rule for indirect jump in Figure 3 was explicitly designed so that the code being jumped to cannot subsequently make recursive calls to itself by an indirect jump through the pointer to itself that is still in the heap. Peter Landin calls this recursion through the heap "tying a knot in the store". A full account of mutual recursion between code and heap types is beyond the scope of this paper, so we only sketch what extensions are needed to accommodate general jumping, what could be typed using them, and how they relate to the type system studied in the previous sections. Typing

$$
\begin{array}{cl}
\Gamma, f \rhd A_1 \Rightarrow_r A_2, \Gamma' \mid \texttt{emp} \vdash \texttt{jmp}\ f : A_1 \Rightarrow_r A_2 & \\
\hline
\Gamma, f \rhd A_1 \Rightarrow_r A_2, \Gamma' \mid \texttt{emp} \vdash \texttt{jmp}\ f : \forall\alpha.(r \mapsto (\forall\alpha_r.(r \mapsto \alpha_r) -\!\!* A_2 -\!\!* \alpha)) -\!\!* A_1 -\!\!* \alpha & \\
\hline
\Gamma, f \rhd A_1 \Rightarrow_r A_2, \Gamma' \mid \texttt{emp} \vdash \texttt{jmp}\ f : (r \mapsto (\forall\alpha_r.(r \mapsto \alpha_r) -\!\!* A_2 -\!\!* A' -\!\!* \alpha')) -\!\!* A_1 -\!\!* A' -\!\!* \alpha' & (\forall\mathrm{E}) \\
\hline
\Gamma, f \rhd A_1 \Rightarrow_r A_2, \Gamma' \mid \texttt{emp} \vdash \texttt{jmp}\ f : (r \mapsto (\forall\alpha_r.(r \mapsto \alpha_r) -\!\!* (A_2 * A') -\!\!* \alpha')) -\!\!* (A_1 * A') -\!\!* \alpha' & (\equiv) \\
\hline
\Gamma, f \rhd A_1 \Rightarrow_r A_2, \Gamma' \mid \texttt{emp} \vdash \texttt{jmp}\ f : \forall\alpha'.(r \mapsto (\forall\alpha_r.(r \mapsto \alpha_r) -\!\!* (A_1 * A') -\!\!* \alpha')) -\!\!* (A_2 * A') -\!\!* \alpha' & (\forall\mathrm{I}) \\
\hline
\Gamma, f \rhd A_1 \Rightarrow_r A_2, \Gamma' \mid \texttt{emp} \vdash \texttt{jmp}\ f : (A_1 * A') \Rightarrow_r (A_2 * A') &
\end{array}
$$

Figure 5. Derivation of a frame law for the function call idiom

$$
\begin{array}{cl}
\hline
\Gamma \mid p \mapsto \forall\alpha_p.((p \mapsto \alpha_p) -\!\!* \alpha) \vdash \texttt{jmp}\ [p] : \alpha & (\textsc{IndJmp}) \\
\hline
\Gamma \mid \texttt{emp} * p \mapsto \forall\alpha_p.((p \mapsto \alpha_p) -\!\!* \alpha) \vdash \texttt{jmp}\ [p] : \alpha & (\equiv) \\
\hline
\Gamma \mid \texttt{emp} \vdash \texttt{jmp}\ [p] : (p \mapsto \forall\alpha_p.((p \mapsto \alpha_p) -\!\!* \alpha)) -\!\!* \alpha & (-\!\!*\mathrm{I}) \\
\hline
\Gamma \mid \texttt{emp} \vdash \texttt{jmp}\ [p] : \forall\alpha.(p \mapsto \forall\alpha_p.((p \mapsto \alpha_p) -\!\!* \alpha)) -\!\!* \alpha & (\forall\mathrm{I}) \\
\hline
\Gamma \mid \texttt{emp} \vdash \texttt{jmp}\ [p] : (p \mapsto \forall\alpha_p.((p \mapsto \alpha_p) -\!\!* A -\!\!* \alpha)) -\!\!* A -\!\!* \alpha & (\forall\mathrm{E}) \\
\hline
\Gamma \mid \texttt{emp} \vdash \texttt{jmp}\ [p] : (p \mapsto \forall\alpha_p.(((p \mapsto \alpha_p) * A) -\!\!* \alpha)) -\!\!* A -\!\!* \alpha & (\equiv) \\
\hline
\Gamma \mid \texttt{emp} * p \mapsto \forall\alpha_p.(((p \mapsto \alpha_p) * A) -\!\!* \alpha) \vdash \texttt{jmp}\ [p] : A -\!\!* \alpha & (-\!\!*\mathrm{E}) \\
\hline
\Gamma \mid \texttt{emp} * (p \mapsto \forall\alpha_p.(((p \mapsto \alpha_p) * A) -\!\!* \alpha)) * A \vdash \texttt{jmp}\ [p] : \alpha & (-\!\!*\mathrm{E}) \\
\hline
\Gamma \mid (p \mapsto \forall\alpha_p.(((p \mapsto \alpha_p) * A) -\!\!* \alpha)) * A \vdash \texttt{jmp}\ [p] : \alpha & (\equiv)
\end{array}
$$

Figure 6. Framing for an indirect jump

knots in the store with a Hoare-style typing is more challenging than in strongly typed functional languages like ML. Since the Hoare typing tracks the state of the heap, it must be different before and after the knot has been tied.

As a concrete example, here is some code in C that ties a knot in the store which then causes the function `f` to loop:

```
void (*p)() = 0;

void f() { (*p)(); }

main()
{
  p = &f;
  f();
}
```

Initially, there is no recursion; `f` only becomes recursive by virtue of the assignment `p = &f;`. If our specification for `f` requires it to be recursive, then we would not be able to call `f` until after the assignment to `p`.

The same principle works with labels in a language with code pointers, like Gnu C, as in the following fragment (using the `&&` operator to turn a label into a pointer):

```
  void *p;

  p = &&f;
  goto f;
  ...
 f: goto *p;
```

Note that `f: goto *p;` does not by itself imply a loop; only the assignment creates the loop.

To type the general form of an indirect jump, we need recursively typed heaps. We write $\mu\phi.A$ for a recursively-defined heap, and assume that we can roll and unroll the recursion via a congruence

$$\mu\phi.A \equiv A[(\mu\phi.A)/\phi]$$

A possible rule for general jumping is then:

$$\frac{}{\Gamma \mid \mu\phi.p \mapsto (\phi -\!\!* C) \vdash \texttt{jmp}\ [p] : C}\ (\mu\textsc{Jmp})$$

Given the operational semantics of $\texttt{jmp}\ [p]$,

$$\langle \texttt{jmp}\ [p] \mid h \mid s \rangle \rightsquigarrow \langle s(h(p)) \mid h \mid s \rangle$$

the precondition

$$\mu\phi.p \mapsto (\phi -\!\!* C)$$

is plausible: it states that the heap contains at address p a pointer to some code that, given the same heap (including p itself), will produce an answer of type C.

Assuming this rule, the code segment consisting of $\{f \mapsto \texttt{jmp}\ [p]\}$ has type

$$f \rhd \mu\phi.p \mapsto (\phi -\!\!* C)) -\!\!* C$$

Assuming this heap, we can then type the code that saves f itself in p and then jumps to f; see Figure 7 for derivations for the code segment containing f and the current code that sets up the knot and then jumps to f. Here C can be any type, in particular $\forall\alpha.\alpha$, which indicates that this code loops (which it does, see Section 2).

A jump in its most general form implies a self-application through the heap, so that it is instructive to compare the derivation in Figure 7 with the typing of the λ-term

$$(\lambda x.xx)\,(\lambda x.xx)$$

using a recursive type $\mu\alpha.(\alpha \to B)$, in particular the unrolling that needs to happen before the self-application xx can be typed.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{- \mid \mu\phi.p \mapsto (\phi \multimap C) \vdash \texttt{jmp}\ [p] : C}\,(\mu\text{JMP})}{- \mid \texttt{emp} * (\mu\phi.p \mapsto (\phi \multimap C)) \vdash \texttt{jmp}\ [p] : C}\,(\equiv)}{- \mid \texttt{emp} \vdash \texttt{jmp}\ [p] : (\mu\phi.p \mapsto (\phi \multimap C)) \multimap C}\,(\multimap\text{I}) \qquad \dfrac{}{\vdash \emptyset : -}\,(\text{EMPTY})}{\vdash \{f \mapsto \texttt{jmp}\ [p]\} : f \triangleright (\mu\phi.p \mapsto (\phi \multimap C)) \multimap C}\,(\text{ADDCODE})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{f \triangleright (\mu\phi.p \mapsto (\phi \multimap C)) \multimap C \mid \texttt{emp} \vdash \texttt{jmp}\ f : (\mu\phi.p \mapsto (\phi \multimap C)) \multimap C}\,(\text{JMP})}{f \triangleright (\mu\phi.p \mapsto (\phi \multimap C)) \multimap C \mid \texttt{emp} * \mu\phi.p \mapsto (\phi \multimap C) \vdash \texttt{jmp}\ f : C}\,(\multimap\text{E})}{f \triangleright (\mu\phi.p \mapsto (\phi \multimap C)) \multimap C \mid \mu\phi.p \mapsto (\phi \multimap C) \vdash \texttt{jmp}\ f : C}\,(\equiv)}{f \triangleright (\mu\phi.p \mapsto (\phi \multimap C)) \multimap C \mid p \mapsto ((\mu\phi.p \mapsto (\phi \multimap C)) \multimap C) \vdash \texttt{jmp}\ f : C}\,(\equiv)}{f \triangleright (\mu\phi.p \mapsto (\phi \multimap C)) \multimap C \mid p \mapsto B \vdash \texttt{movc}\ f\ p; \texttt{jmp}\ f : C}\,(\text{MOVCODE})$$

Figure 7. Recursion by tying a knot in the heap

Recall the restricted idiom of jumping that avoids recursion, as given in Figure 3 and discussed in Section 3.1. This non-recursive indirect jump rule has the precondition

$$p \mapsto (\forall\alpha_p.(p \mapsto \alpha_p) \multimap C)$$

The general recursive jump rule has the precondition

$$\mu\phi.p \mapsto (\phi \multimap C)$$

We can relate the latter to the former by instantiating α_p to $(\mu\phi.(p \mapsto (\phi \multimap C))) \multimap C$, and rolling the recursive type twice, as follows:

$$\begin{array}{ll} & p \mapsto (\forall\alpha_p.(p \mapsto \alpha_p) \multimap C) \\ & p \mapsto (p \mapsto ((\mu\phi.(p \mapsto (\phi \multimap C))) \multimap C) \multimap C) \\ = & p \mapsto ((\mu\phi.(p \mapsto (\phi \multimap C))) \multimap C) \\ = & \mu\phi.(p \mapsto (\phi \multimap C)) \end{array}$$

The operational semantics of indirect jumping remains the same, but what is different is the view of what is passed along with the jump.

To summarize, it appears that, subject to the requirement for contravariant recursive types, recursion through dynamically created knots in the heap would fit quite smoothly into the present type system.

8. Conclusions

We have recovered frame rules from answer type polymorphism over $\perp\!\perp$-closed types. The potential recursion through code pointers in the heap remains a challenge, but may be amenable to similar techniques.

8.1 Related work

Work on typed assembly language [16], and typing heaps, such as substructural type systems [18], is part of the general background of the present paper. Recent work on L3, a Linear Logic with Locations [17], also uses a relational style of semantics, rather than subject reduction, for soundness of a type system for a language with heap operations. Type systems and logics for assembly languages usually assume more restricted control flow than pointers to unknown code; however, in recent work Ni and Shao have studied a language with embedded code pointers [20]. Much of the typing presented here is based on separation logic and bunched typing [21, 24] in particular. Code has to work not just given the currently known code segment, but all larger ones, to which it may gain access through code pointers, so it appears the semantics could be formulated as a possible-worlds semantics [25].

Apart from separation logic, substructural type systems and answer type polymorphism, one of the crucial ideas is biorthogonality. It was invented by Krivine [11] and independently by Pitts, who defines a notion of $\top\top$-closure of relations [23] in his relational parametricity for operational semantics. Vouillon and Melliés have recently built on Krivine's work [15, 31] by giving elegantly simple semantics for polymorphic and recursive types in an operational setting, as an alternative to approaches based on indexing [1]. Lindley and Stark [13] use Pitts's $\top\top$-closure for termination proofs.

Both in Krivine's and Pitts's version, orthogonality is a relation between a term and its continuation, which is syntactically represented as a stack of frames [7]. (Krivine also considers operating systems-level features such as the run-time clock [12] in realizability.) In the present paper, we transfer the idea of orthogonality to assembly language, where there is no surrounding evaluation context; rather the continuation in this sense is the rest of the heap and code segment, and spatial separation is built into the definition of orthogonality.

8.2 Directions for further work

It seems quite likely that Melliés's and Vouillon's recent work on recursive types in an operational setting [15] could be adopted to deal with the recursive types that are necessary to cover indirect jumps in full generality, as in Section 7. In fact, this possible direction is one of the reasons why a framework inspired by their use of biorthogonality was used in this paper. Principled operational techniques are also appropriate in this context since the literature on low-level languages typically presents them in terms of operational semantics. Conversely, the mutual recursion between code (which operates on heaps) and heaps (which can point to code) provides concrete motivation for recursive types combined with polymorphism.

Frame rules via answer type polymorphism should generalize from function calls to more general forms of jumping. It remains to be seen if it can cover the hypothetical frame rule [22] or higher-order frame rules, perhaps even for non-functional control structure such as coroutines or system calls. BI-style typing of continuations may also overcome some of the limitations of earlier work on linear continuation passing [3, 5].

Multiplicative quantification over locations may be an alternative to biorthogonality for obtaining well-behaved answer type polymorphism [4]. The distinction between an immutable code segment and a writable heap could be seen as an instance of permissions [6] in separation logic, in that $f \triangleright B$ corresponds to an execute permission and $f \mapsto B$ to a read and write permission in operating systems. By using execute permissions, it may be possible to unify the code segment and the heap to give greater flexibility than in the system presented here, for instance for dynamically loading code.

Acknowledgments

Thanks to Josh Berdine, Cristiano Calcagno and Peter O'Hearn for discussions, and to the anonymous POPL referees for comments.

References

[1] Andrew A. Appel and David McAllester. An indexed model for recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(5):657–683, 2001.

[2] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings 27th Principles of Programming Languages (POPL '00)*, pages 243–253. ACM, 2000.

[3] Josh Berdine. *Linear Typing of Continuation Passing Style*. PhD thesis, Queen Mary, University of London, 2002.

[4] Josh Berdine and Peter W. O'Hearn. Strong update, disposal and encapsulation in bunched typing. Draft, October 2005.

[5] Josh Berdine, Peter W. O'Hearn, Uday Reddy, and Hayo Thielecke. Linear continuation passing. *Higher-order and Symbolic Computation*, 15(2/3):181–208, 2002.

[6] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In *Principles of Programming Languages (POPL'05)*, pages 259–270. ACM, 2005.

[7] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts*, pages 193–217. North-Holland, 1986.

[8] Timothy G. Griffin. A formulae-as-types notion of control. In *Principles of Programming Languages (POPL '90)*, pages 47–58. ACM, 1990.

[9] Samin S. Ishtiaq and Peter O'Hearn. BI as an assertion language for mutable data structures. In *Principles of Programming Languages (POPL '01)*, pages 14–26. ACM, 2001.

[10] Pierre Jouvelot and David K. Gifford. Reasoning about continuations with control effects. In *Programming Language Design and Implementation (PLDI)*, pages 218–226. ACM, 1988.

[11] Jean-Louis Krivine. Typed lambda-calculus in classical Zermelo-Fraenkel set theory. *Archive of Mathematical Logic*, 40(3):189–205, 2001.

[12] Jean-Louis Krivine. Dependent choice, 'quote' and the clock. *Theoretical Computer Science*, 308(1–3):259–276, 2003.

[13] Sam Lindley and Ian Stark. Reducibility and TT-lifting for computation types. In *Typed lambda calculus and applications (TLCA)*, number 3461 in LNCS, pages 262–277. Springer, 2005.

[14] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Principles of Programming Languages (POPL '88)*, pages 47–57. ACM, 1988.

[15] Paul-André Melliés and Jérôme Vouillon. Recursive polymorphic types and parametricity in an operational framework. In *Logic and Computer Science (LICS'05)*. IEEE, 2005.

[16] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In *Principles of Programming Languages (POPL '98)*, pages 85–97. ACM, 1998.

[17] Gregory Morrisett, Amal J. Ahmed, and Matthew Fluet. L3: A linear language with locations. In *Typed Lambda Calculus and Applications (TLCA)*, volume 3461, pages 293–307. Springer, 2005.

[18] Gregory Morrisett, F. Smith, and D. Walker. Alias types. In *Proceedings European Symposium on Programming (ESOP)*, volume 1782 of *LNCS*, pages 366–381. Springer, 2000.

[19] George C. Necula. Proof-carrying code. In *Principles of Programming Languages (POPL '97)*, pages 106–119. ACM, 1997.

[20] Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In *Proc. 33rd ACM Symposium on Principles of Programming Languages (POPL'06)*. ACM, January 2006. (to appear).

[21] Peter W. O'Hearn. On bunched typing. *Journal of Functional Programming*, 13(4):747–796, 2003.

[22] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *Principles of Programming Languages (POPL'04)*, pages 268–280, 2004.

[23] Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.

[24] David J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*. Kluwer Academic Publishers, 2002.

[25] David J. Pym, Peter W. O'Hearn, and Hongseok Yang. Possible worlds and resources: the semantics of BI. *Theoretical Computer Science*, 315(1):257–305, 2004.

[26] John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).

[27] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, pages 55–74. IEEE, 2002.

[28] Hayo Thielecke. Comparing control constructs by double-barrelled CPS. *Higher-order and Symbolic Computation*, 15(2/3):141–160, 2002.

[29] Hayo Thielecke. From control effects to typed continuation passing. In *30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, pages 139–149. ACM, 2003.

[30] Hayo Thielecke. Answer type polymorphism in call-by-name continuation passing. In *European Symposium on Programming (ESOP 2004)*, volume 2986 of *LNCS*, pages 279–293. Springer, 2004.

[31] Jérôme Vouillon and Paul-André Melliés. Semantic types: a fresh look at the ideal model for types. In *Principles of Programming Languages (POPL'04)*, pages 52–63, 2004.

[32] Philip Wadler. Theorems for free! In *4'th International Conference on Functional Programming and Computer Architecture (FPCA'89)*, pages 347–359. ACM, 1989.