Singleton

A General-Purpose Dependently-Typed Assembly Language

Simon Winwood
NICTA and UNSW
sjw@cse.unsw.edu.au

Manuel Chakravarty
UNSW
chak@cse.unsw.edu.au

Abstract

In this paper we present Singleton, a dependently typed assembly language. Based upon the calculus of inductive constructions, Singleton's type system allows procedures abstracting over terms, types, propositions, and proof terms.

Furthermore, Singleton includes generalised singleton types. In addition to the primitive singleton types of other languages, these generalised singleton types allow the values from arbitrary inductive types to be associated with the contents of registers and memory locations. Along with Singleton's facility for term and proof abstraction, generalised singleton types allow strong statements to be made about the functional behaviour of Singleton programs.

We have formalised basic properties of Singleton's type system, namely type safety and a type erasure property, using the Coq proof assistant.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Logics of programs

General Terms Theory

1. Introduction

The Singleton language attempts to address a gap in the available languages for certificate-bearing code. Morrisett *et al.* [9] gives an elegant translation from a high level language — System F — into a TAL, thus showing that typed assembly languages are natural certification logics for programs compiled from high-level languages. Furthermore, the guarantees provided by the type system of a TAL typically correspond with the low-level safety properties desired of the system: well typed programs cannot 'go wrong'.

However, traditional TALs are restricted to those properties entailed by type safety. While this includes essential properties such as memory safety, these properties are not always sufficient. For instance, while traditional TALs may be able to encode that a particular address contains the root of a tree, they cannot state that the tree is balanced.

Based on the *calculus of inductive constructions* (CiC), Singleton's assertion logic allows programs to abstract over terms in this logic, including types, terms, propositions, and proofs. Singleton also includes a rich type language including *generalised singleton types*; these types carry terms in the assertion logic corresponding to the run-time behaviour of the classified object.

Example 1.1 As an illustrative example of the power of Singleton's type system, consider the higher-order function map with type

$$map \ : \ \forall a \ b. \ (a \ \rightarrow \ b) \ \rightarrow \ list \ a \ \rightarrow \ list \ b$$

A traditional TAL may admit an implementation of such a function, but Singleton allows a program to prove that the output is actually the result of applying a given function to every element of the input list; the Singleton type

$$\begin{aligned}\texttt{map} \ :: \ &\forall(a \ : \ Set)(b \ : \ Set)(f \ : \ a \ \rightarrow \ b)(xs \ : \ list \ a). \\ &\{\texttt{a}_1 \ : \ \forall(v \ : \ a).\{\texttt{a}_1 \ : \ sgl(v : a), \\ &\qquad\qquad \texttt{ra} \ : \ \{\texttt{t}_1 \ : \ sgl(f \ v : b)\}\}, \\ &\ \ \texttt{a}_2 \ : \ sgl(xs : list \ a), \\ &\ \ \texttt{ra} \ : \ \{\texttt{t}_1 \ : \ sgl(map \ f \ xs : list \ b)\}\}\end{aligned}$$

states that the `map` program takes for arguments the logical types a and b, a function f from a to b, and a list with element type a. The program also takes as run-time arguments a function pointer in $\texttt{a}_1$ which computes fv for any v, a generalised singleton object in $\texttt{a}_2$ representing the input list, and a return address in `ra` expecting as argument $map \ f \ xs$: if the program returns through this address, it must provide a generalised singleton object representing $map \ f \ xs$, that is, the contents of $\texttt{t}_1$ will correspond to the result of mapping f over the elements in xs.

Our original motivation in developing Singleton was in the context of run-time verification: we wished to establish high-level properties of the program which are not implied by type safety alone. In fact, the requirements of run-time verification are more comparable to those of a traditional verification environment such as a Hoare-style logic [8].

Such languages, however, typically assume a much looser view of type safety than run-time verification requires; for example, a reference monitor is usually type safe, and will not need to modify arbitrary memory locations.

Singleton is then a compromise between the simplicity of a typed assembly language and the expressive power of a Hoare logic. The use of generalised singleton types, types which carry detailed information about their run-time contents, allows fine-grained assertions on heap structures; the addition of existential types allows control over the mutability of objects in the heap. Other languages, the DTAL of Xi and Harper [15] for example, typically provide singletons on primitive types, such as the value of words or the length of arrays; Singleton allows user-defined singleton types over inductive data types.

In summary, the main contributions of this paper are the following

TLDI'11, January 25, 2011, Austin, Texas, USA.

```
head :: ∀(a : Set)(xs : list a).
          {a1 : sgl(xs : list a),
           ra : ∀(x : list a)(xs′ : list a)
                 (pf : xs = cons x xs′).{t1 : sgl(x : a)}}
head :
    case t1, a1, [hNil @ (a, xs), ]
                  hCons @ (a, xs)

hNil :: ∀(a : Set)(xs : list a)(pf : xs = nil).
          {t1 : ⟨word(0)⟩,
           ra : ∀(x : list a)(xs′ : list a)
                 (pf : xs = cons x xs′).{t1 : sgl(x : a)}}
hNil :
    (handle error for empty list)

hCons :: ∀(a : Set)(xs : list a)(x : a)(xs′ : list a)
           (pf : xs = cons x xs′).
           {t1 : ⟨word(1), sgl(x : a), sgl(xs′ : list a)⟩,
            ra : ∀(x : list a)(xs′ : list a)
                  (pf : xs = cons x xs′).{t1 : sgl(x : a)}}
hCons :
    load t1, t1(1)
    apply ra, ra, (x, xs′, pf)
    jump ra
```

Figure 1. A Singleton implentation of the *head* function. The argument register $\mathtt{a}_1$ contains the input list.

- We introduce the idea of *generalised singleton types*, that enable user-defined singleton types corresponding to arbitrary inductive data types (Section 2).
- We present Singleton, a dependently-typed assembly language, that includes an assertion logic based on the calculus of inductive constructions (Sections 3 and 4)
- We give an operational and a static semantics for Singleton (Sections 5 and 6).
- We establish fundamental properties of Singleton's type system; refer for proofs to [14] (Section 7)

We will discuss related work in more detail in Section 8.

2. Singleton by Example

We give two short examples to introduce Singleton and also to serve as a running example through the remainder of this paper.

2.1 Taking the head of a list

Consider the list data type

$$\begin{array}{l}\textbf{Inductive}\ \mathit{list}\ (t\ :\ \mathit{Set})\ :\ \mathit{Set}\ :=\\ \quad |\ \mathit{nil}\ :\ \mathit{list}\ t\\ \quad |\ \mathit{cons}\ :\ t\ \rightarrow\ \mathit{list}\ t\ \rightarrow\ \mathit{list}\ t\end{array}$$

and the *head* function

$$\begin{array}{l}\mathit{head}\ \ :: \forall a.\ \mathit{list}\ a\ \rightarrow\ a\\ \mathit{head}\ \mathit{xs} = \textbf{match}\ \mathit{xs}\ \textbf{with}\\ \qquad\qquad \mathit{nil}\ \rightarrow\ \mathit{error}\\ \qquad\qquad \mathit{cons}\ x\ \mathit{xs}'\ \rightarrow\ x\\ \qquad\quad \textbf{end}\end{array}$$

where *error* indicates a runtime error when the list is empty. We can, of course, make this function total by requiring a proof that xs isn't nil; see Sect. 2.1.4. Fig. 1 gives the Singleton implementation of this function.

2.1.1 The two types

We claim that the Singleton implementation is somehow related to *head*. To see why we can make this claim, consider the type of *head*

$$\mathit{head}\ ::\ \forall a.\ \mathit{list}\ a\ \rightarrow\ a$$

and that for `head`

$$\begin{array}{l}\forall(a\ :\ \mathit{Set})(\mathit{xs}\ :\ \mathit{list}\ a).\\ \quad\{\mathtt{a}_1\ :\ \mathit{sgl}(\mathit{xs} : \mathit{list}\ a),\\ \quad\ \mathtt{ra}\ :\ \forall(x\ :\ \mathit{list}\ a)(\mathit{xs}'\ :\ \mathit{list}\ a)\\ \qquad\quad (\mathit{pf}\ :\ \mathit{xs}\ =\ \mathit{cons}\ x\ \mathit{xs}').\{\mathtt{t}_1\ :\ \mathit{sgl}(x : a)\}\}\end{array}$$

We can read this type as a code block which takes the *logical* arguments a and xs, and two *run-time* arguments in the registers $\mathtt{a}_1$ and `ra`. The register `ra` contains a reference to a code block with logical arguments x, xs', and pf, and a single register argument, $\mathtt{t}_1$.

Both the type of *head* and `head` are polymorphic in the contents of the list; this is the logical parameter a in the Singleton version. Similarly, both accept the input list, although in the Singleton version this occurs both as the logical parameter xs and as the register argument in $\mathtt{a}_1$: the type

$$\mathit{sgl}(\mathit{xs} : \mathit{list}\ a)$$

connects the logical variable xs to the machine register $\mathtt{a}_1$; we discuss what this means in the next section.

Finally, both return a value of type a. In the Singleton case, this can be seen by examining the type of the return address: if `head` returns through this address, it will provide the logical parameters x and xs' representing the head and tail of the list respectively, and a *proof* that xs is, in fact, composed of these two values. As with the argument to `head`, the head of the list, x, appears as the register argument $\mathtt{t}_1$.

Therefore, we claim that if `head` returns through the given return address, then the result in $\mathtt{t}_1$ will correspond to the head of the list xs.

2.1.2 Singleton types

In the previous section we noted that the register $\mathtt{a}_1$ somehow contains the list xs, and the register $\mathtt{t}_1$ should contain the value x when `head` returns. We now examine more closely the types of these two registers to see precisely how the argument and result of *head* correspond to their Singleton counterparts.

Recall the list argument to `head`: the *generalised singleton type* $\mathit{sgl}(\mathit{xs} : \mathit{list}\ a)$ represents the logical value xs with logical type $\mathit{list}\ a$; we say that xs is the (logical) value associated with the singleton type, and similarly for its type, $\mathit{list}\ a$.

Objects with singleton type have an underlying object, the type of which, the *representation type*, is determined by the value associated with the singleton. How this type is determined in general we leave to Sect. 6.3; here we will focus on the details required for *head*.

We start by noting that xs has two possible forms: either it is nil, or it is $\mathit{cons}\ x\ \mathit{xs}'$ for some x and xs'. In the first case, $\mathit{sgl}(\mathit{nil} : \mathit{list}\ a)$ is represented by the tuple

$$\langle \mathit{word}(0)\rangle$$

that is, the tuple containing the singleton word 0 indicating a nil node; in the second case, $\mathit{sgl}(\mathit{cons}\ x\ \mathit{xs}' : \mathit{list}\ a)$ is represented by the tuple

$$\langle \mathit{word}(1),\ \mathit{sgl}(x : a),\ \mathit{sgl}(\mathit{xs} : \mathit{list}\ a)\rangle$$

that is, the word 1 indicating a $cons$ node, followed by an element for x and then an element for xs'.

Thus, depending on the contents of xs, the register $\mathtt{a}_1$ may contain a tuple with a single member, or a tuple with three members. We can then say that the argument in $\mathtt{a}_1$ corresponds to the argument to $head$, xs, in that the first element of xs is the second entry in the tuple in $\mathtt{a}_1$, the second element of xs is second entry of the tuple in third entry of this tuple, and so on.

2.1.3 The implementation of `head`

We now turn our attention to the actual implementation of `head`. As noted above, the value associated with a singleton determines the shape of the representation object. Conversely, by examining this object, specifically the first element, we can determine the shape of the associated value: there is a one-to-one correspondence between the first element of a representation object and the associated value.

Thus, by examining the first element of a list singleton we can determine whether it is nil or $cons\ x\ xs'$ for some x and xs'. In fact, this is precisely the behaviour of `case`: the instruction

$$\mathtt{case}\ \mathtt{t}_1,\ \mathtt{a}_1,\ [\mathtt{hNil} \mathbin{@} (a,\ xs),\ \mathtt{hCons} \mathbin{@} (a,\ xs)]$$

examines the singleton value in $\mathtt{a}_1$ and branches to `hNil` if the value associated with $\mathtt{a}_1$ is nil and to `hCons` if the value is $cons$. The unpacked singleton value is passed in $\mathtt{t}_1$. In both cases we pass the additional logical arguments a and xs.

In addition to these explicit arguments, the `case` instruction passes to the target label any constructor arguments, along with a proof. This proof states that these additional arguments, applied to the corresponding constructor, give the associated value.

The logical arguments to `hNil` are then the two explicit arguments followed by a proof that the list is indeed empty. As this case is undefined, and will result in some implementation-specific error, we shall treat it no further.

The `hCons` case, however, is more interesting. In addition to the element type and argument list, this code block also accepts the two arguments to the $cons$ constructor, x and xs', along with a proof that these do actually form xs, namely

$$pf\ :\ xs\ =\ cons\ x\ xs'$$

Furthermore, the type of register $\mathtt{t}_1$ is the representation type for $cons\ x\ xs'$, that is, a tuple containing the tag 1, an element for x, and an element for xs'. We then load the second entry to get x.

Before we can return to the address in `ra` we need to communicate to this label the new variables and the new proof; this is done via the instruction

$$\mathtt{apply}\ \mathtt{ra},\ \mathtt{ra},\ (x,\ xs',\ pf)$$

where $(x,\ xs',\ pf)$ is an *argument sequence*, that is, a sequence of logical arguments and a sequence of type arguments, although in this case we have no type arguments. This instruction applies the arguments in this sequence to the code type in `ra`. After applying the new variables, we jump to the resulting address.

2.1.4 Aside: a total version of $head$

Our version of $head$ above raises an error in the case that xs is nil. If we know that xs is *not* nil, that is, we have $xs\ =\ cons\ x\ xs'$, then we can give a total version of `head` as follows

$$\begin{array}{l}
\mathtt{totalHead}\ ::\ \forall(a : Set)(xs : list\ a)(x : list\ a)(xs' : list\ a) \\
\qquad\qquad (pf : xs = cons\ x\ xs'). \\
\qquad\qquad \{\mathtt{a}_1 : sgl(xs : list\ a), \\
\qquad\qquad\ \ \mathtt{ra} : \{\mathtt{t}_1 : sgl(x : a)\}\} \\
\mathtt{totalHead}: \\
\quad \mathtt{coerce}\ \mathtt{t}_1,\ \mathtt{a}_1\ \mathtt{using}\ pf \\
\quad \mathtt{project}\ \mathtt{t}_1,\ \mathtt{t}_1 \\
\quad \mathtt{load}\ \mathtt{t}_1,\ \mathtt{t}_1(1) \\
\quad \mathtt{jump}\ \mathtt{ra}
\end{array}$$

We take advantage of two Singleton-specific instructions in this implementation of $head$, `coerce` and `project`. Firstly, the instruction

$$\mathtt{coerce}\ r_d,\ r_s\ \mathtt{using}\ pf$$

rewrites the value associated with the singleton in r_s with the equality in pf, placing the result in r_d. In our example above, we have

$$sgl(xs : list\ a) \xrightarrow{rewrite\ with\ pf} sgl(cons\ x\ xs' : list\ a)$$

Secondly, the instruction

$$\mathtt{project}\ r_d,\ r_s$$

extracts the representation of the singleton in r_s into r_d, when the shape of the associated value is known. In the above case, we have

$$sgl(cons\ x\ xs' : list\ a) \xrightarrow{project} \langle word(1),\ sgl(x : a),\ sgl(xs' : list\ a)\rangle$$

noting that the destination register of `project` will have the same type as if after a `case` operation. This instruction is thus an optimised version of `case`: `project` simply extracts the representation type for a *known* value, while `case` examines an *unknown value* and branches to the appropriate label.

2.2 Adding an element to a list

We have seen how to destroy a singleton object: use `case` if the shape of the associated value is unknown, and `project` if the value is known. In this section we will discuss how to *construct* a singleton.

Continuing with our example of lists, the `cons` procedure should act like the $cons$ constructor, that is, it should add the given element to the start of the given list.

If we wish to create a new singleton object, we use the `inject` instruction. This operation performs the inverse of `project`: given an object of the correct type, it creates a singleton such that the object represents the singleton. Recall from Sect. 2.1.2 that a $cons$ term is represented by a triple: the tag 1, the head of the list, and the tail of the list.

Given the head of the list in $\mathtt{a}_1$ and the tail in $\mathtt{a}_2$, our implementation needs to allocate a tuple and initialise the first element with the word 1 and the remainder with the contents of $\mathtt{a}_1$ and $\mathtt{a}_2$. The code is then as follows

$$\begin{array}{l}
\mathtt{cons}\ ::\ \forall(a : Set)(x : a)(xs : list\ a). \\
\qquad\qquad \{\mathtt{a}_1 : sgl(x : a), \\
\qquad\qquad\ \ \mathtt{a}_2 : sgl(xs : list\ a), \\
\qquad\qquad\ \ \mathtt{ra} : \{\mathtt{t}_1 : sgl(cons\ x\ xs : list\ a)\}\} \\
\mathtt{cons}: \\
\quad \mathtt{alloc}\ \mathtt{t}_1,\ [word(1), sgl(x : a), sgl(xs : list\ a)] \\
\quad \mathtt{ldi}\ \mathtt{t}_2,\ 1 \\
\quad \mathtt{store}\ \mathtt{t}_2,\ \mathtt{t}_1(0) \\
\quad \mathtt{store}\ \mathtt{a}_1,\ \mathtt{t}_1(1) \\
\quad \mathtt{store}\ \mathtt{a}_2,\ \mathtt{t}_1(2) \\
\quad \mathtt{inject}\ \mathtt{t}_1,\ \mathtt{t}_1\ \mathtt{as}\ sgl(cons\ x\ xs : list\ a) \\
\quad \mathtt{jump}\ \mathtt{ra}
\end{array}$$

The `alloc` instruction creates a new, uninitialised tuple with the given type in $\mathtt{t}_1$. We then use the `ldi` instruction to move the constant 1 into $\mathtt{t}_2$ so we can initialise the first element of the new tuple. After storing the two arguments, we create the new singleton using the `inject` instruction. We then return to the given address.

$$
\begin{array}{lcl}
A,\ B,\ v,\ T & ::= & \mathit{Set} \mid \mathit{Prop} \mid \mathit{Type} \\
& \mid & x \mid \lambda x : A.\ B \mid A\ B \mid \forall x : A.\ B \\
& \mid & \mathit{Ind}(X : A)\{\tilde{B}\} \\
& \mid & \mathit{Ctor}(n,\ A) \\
& \mid & \mathit{Elim}(C,\ A)\{\tilde{B}\} \\
\Upsilon & ::= & x_1 : A_1,\ \ldots,\ x_n : A_n
\end{array}
$$

Figure 2. The calculus of inductive constructions, used by Singleton as an assertion logic.

2.3 Discussion

The natural question to ask is: what have we achieved by the extra type machinery involved in a (type-correct) Singleton program.

Firstly, if the return address is invoked, the contents of the register $\mathtt{t}_1$ will have the singleton type $sgl(x : a)$; Singleton provides a logic of *partial* correctness only, and so the `head` function may not return through the given label (calling some other function, for example) or may not return at all.

Secondly, objects with singleton type have a specific layout: given the type, we can make strong statements about the contents of values of this type in the heap.

Finally, any object in the heap retains the type it had before the invocation of `head`. Although this is not the same as saying that the remainder of the heap is unmodified, Singleton's precise types imply that either such modifications are trivial (that is, equivalent to the identity mutation), or the element modified had existential type and therefore any mutation does not violate invariants about the particular value of that cell.

3. The Calculus of Inductive Constructions

In this section we summarise the syntax of CiC and briefly justify its use. Space limitations prevent us from giving a complete treatment of CiC; we direct the reader to Paulin-Mohring [11].

The syntax of CiC is given in Fig. 2. Apart from sorts (Set, Prop, and Type) and the usual lambda terms, application, and (dependent) function spaces, CiC includes inductive datatypes ($\mathit{Ind}(X{:}A)\{\vec{B}\}$), along with their inhabitants ($\mathit{Ctor}(n, A)$). Elimination ($\mathit{Elim}(C, A)\{\vec{B}\}$) takes apart inductive terms, generalising recursive functions and structural induction. Following the Coq system [2], we will use a concrete syntax for inductive types, like that above for the definition of list.

The CiC has a number of features which make it a good fit as an assertion logic for Singleton, namely

- dependent types allow a uniform syntax for both terms and proofs, including their abstraction;
- inductive types give structure to singleton values. A primitive notion of an inductive type allows Singleton to give rules for automatically generating this structure;
- separating informative types (that is, those in Set) from non-informative types (that is, those in Prop) allows the values associated with singleton types to include proof objects without a run-time penalty; and
- the construction of a formal model of Singleton in the Coq system requires no extra machinery for the assertion logic beyond that provided by the system.

We shall revisit these first two points in the remainder of this paper, particularly Sect. 6.3. We direct interested in the final point to the first author's dissertation [14].

The typing judgement $\Upsilon \Vdash A : B$ states that the term A has type B under the context Υ, while the equivalence relation $A \equiv_{\beta\iota} B$ states that the two terms A and B are equivalent under β- and iota-reduction (elimination of inductive terms). Both typing and term equivalence are decidable.

$$
\begin{array}{lcl}
\textit{Initialisation flags} & & \\
\varphi & ::= & \mathbf{0} \mid \mathbf{1} \\
\textit{Registers} & & \\
r & ::= & \mathtt{r}_1 \mid \ldots \mid \mathtt{r}_N \\
\\
\textit{Kinds and machine types} & & \\
\kappa,\ \pi & ::= & \mathcal{U} \mid \mathcal{B} \\
\sigma,\ \tau,\ \xi & ::= & \alpha \\
& \mid & \mathit{word}(n) \\
& \mid & \mathit{sgl}(v : T) \\
& \mid & \exists (x : A).\tau \\
& \mid & \langle \tau_1^{\varphi_1},\ \ldots,\ \tau_n^{\varphi_n} \rangle \\
& \mid & \forall \Upsilon, [\Delta].\Gamma \\
\textit{Type contexts} & & \\
\Delta,\ \Omega & ::= & \{\alpha_1 :: \kappa_1,\ \ldots,\ \alpha_n :: \kappa_n\} \\
\textit{Register file types} & & \\
\Gamma & ::= & \{\mathtt{r}_0 : \tau_1,\ \ldots,\ \mathtt{r}_n : \tau_n\} \\
\textit{Heap types} & & \\
\Psi & ::= & \{0 : \tau_1,\ \ldots,\ n : \tau_n\}
\end{array}
$$

Figure 3. Syntactic classes for Singleton: types

4. Syntax

In this section we present the syntax of Singleton, and discuss some of the syntactic forms; the remainder are discussed in the following sections. The various syntactic forms constituting the Singleton language are shown in Fig. 3 and Fig. 4.

4.1 Types and Values

In this section we discuss the various types supported by Singleton, along with their value forms. Briefly, kinds classify types into *unboxed* and *boxed* types, while types classify both word values and heap values. Heap values are code blocks and tuples of word values, while word values appear in registers and in tuples. In the remainder of this section we will discuss types and their corresponding values.

Type variables. Type variables refer to variables bound by code types. We assume α-equivalence on type variables, renaming where necessary to avoid capture.

Word types. Word types represent the exact run-time value of their members. For example, a register with type $\mathit{word}(3)$ will contain the integer[1] 3.

The values classified by word types are simply the natural numbers, $\mathbb{N}$.

Generalised singletons. Singleton types carry an abstract representation of their run-time value as a logical term; the singleton type $\mathit{sgl}(v : T)$ classifies objects which correspond to the logical value v with (logical) type T.

This correspondence between an object of singleton type and the associated value is through that value's *representation type*, a machine type derived from the constructor and arguments which form the value; this derivation is discussed in Sect. 6.3.

A singleton value has the form $\mathit{sgl}\ (v :\ T)\ \mathit{in}\ wv$ where v and T are the value and type associated with the singleton. The word value wv is a reference to the *representation object* for the singleton, that is, wv should have the representation type derived from v; we shall discuss the structure of this object in Sect. 6.3 with the representation type.

[1] For simplicity, we assume unbounded machine words

Arguments sequences and extended labels

$$\begin{array}{lcl} \Sigma & ::= & (\tilde{A};\ \tilde{\tau}) \\ \mathcal{L} & ::= & \mathbb{N} \mathbin{@} \Sigma \end{array}$$

Instructions and instruction sequences

$$\begin{array}{lcl} I & ::= & \texttt{add}\ r_d,\ r_1,\ r_2 \\ & \mid & \texttt{ldi}\ r,\ n \\ & \mid & \texttt{move}\ r_d,\ r_s \\ & \mid & \texttt{beq}\ r_1,\ r_2,\ \mathcal{L}\ \texttt{as}\ x \\ & \mid & \texttt{apply}\ r_d,\ r_s,\ \Sigma \\ & \mid & \texttt{lda}\ r,\ \mathcal{L} \\ & \mid & \texttt{load}\ r_d,\ r_s(n)\ \mid \texttt{store}\ r_d,\ r_s(n) \\ & \mid & \texttt{alloc}\ r_d,\ \tilde{\tau} \\ & \mid & \texttt{pack}\ r_d,\ r_s\ \texttt{as}\ \tau\ \texttt{hiding}\ v \\ & \mid & \texttt{unpack}\ r_d,\ r_s\ \texttt{as}\ x \\ & \mid & \texttt{inject}\ r_d,\ r_s\ \texttt{as}\ \tau \\ & \mid & \texttt{project}\ r_d,\ r_s \\ & \mid & \texttt{coerce}\ r_d,\ r_s\ \texttt{using}\ p \\ IS & ::= & I\ ;\ IS \\ & \mid & \texttt{case}\ r_d,\ r_s,\ \tilde{\mathcal{L}} \\ & \mid & \texttt{br}\ \mathcal{L} \mid \texttt{jump}\ r \mid \texttt{halt}\ [\tau] \end{array}$$

Word values

$$wv \quad ::= \quad ?\tau \mid \mathbb{N} \mid \mathcal{L} \mid \mathit{pack}\ v\ \mathit{in}\ wv \mid \mathit{sgl}\ (v :\ T)\ \mathit{in}\ wv$$

Heap values

$$hv \quad ::= \quad \langle wv_1, \ldots, wv_n \rangle \mid \Lambda\, \Upsilon, [\Delta].IS$$

Heaps, register files, and programs

$$\begin{array}{lcl} H & ::= & \{0 \mapsto hv_1,\ \ldots,\ n \mapsto hv_n\} \\ R & ::= & \{\mathtt{r}_0 \mapsto wv_1,\ \ldots,\ \mathtt{r}_N \mapsto wv_N\} \\ P & ::= & (H,\ R,\ IS) \end{array}$$

Figure 4. Syntactic classes for Singleton: instructions and values

Existential types. Singleton and word types are sometimes *too* precise: we may not care about the particular value associated with a type, only its general shape. The existential type $\exists(x : T).\tau$ then hides a logical value inside the type τ. In particular, an existential can hide the value associated with a singleton type.

Example 4.1 We can implement the non-dependent word type by

$$WORD \triangleq \exists(n : nat).word(n).$$

that is, a word type where the associated value is hidden by an existential.

Existential types are not limited to hiding logical values (that is, objects with a type in the sort Set): other logical terms can be hidden, including proof terms[2].

Example 4.2 The type

$$\exists(b\ :\ nat)(p\ :\ \mathit{if}\ b\ >\ 0\ \mathit{then}\ P\ \mathit{else}\ Q).word(b)$$

that is, a word type with a hidden value along with a (hidden) proof, can be used to simulate a boolean type. Depending on the value of b, p is equivalent to a proof of the proposition P or a proof of Q.

Tuple types. Tuple types are sequences of types classifying sequences of corresponding word values in the heap. Tuple types also track the initialisation status of the tuple contents: each type in the sequence has an associated *initialisation flag* which tracks the initialisation status of that entry [9]. For clarity, the initialisation flag may be omitted if it is set.

Values of tuple type occur both as heap values and as word values; in the heap, a tuple type classifies a sequence of word values, while as a word value, a tuple type classifies a label. Such labels then refer to a sequence of values in the heap.

Code types. Program blocks may abstract over both types and logical terms. The code type $\forall\, \Upsilon, [\Delta].\Gamma$ classifies code blocks abstracting over the logical variables in Υ and the type variables in Δ. The term Γ gives the expected types of each register; not all registers need to have a type.

Example 4.3 The type of a function which doubles the word in $\mathtt{t}_1$ and preserves the word in $\mathtt{t}_2$, returning to the label in `ra` is

$$\begin{array}{l} \forall(x : nat), [\alpha :: \mathcal{B}].\{\mathtt{t}_1 : word(x), \mathtt{t}_2 : \alpha, \\ \qquad\qquad\qquad\qquad \mathtt{ra} : \{\mathtt{t}_1 : word(x + x), \mathtt{t}_2 : \alpha\} \end{array}$$

where the logical argument is $(x\ :\ nat)$, the type argument is α, and the register arguments are $\mathtt{t}_1$ with word type $word(x)$, $\mathtt{t}_2$ with parametric type α, and the return register `ra` with the code type $\{\mathtt{r}_1\ :\ word(x\ +\ x),\ \mathtt{r}_2\ :\ \alpha\}$.

Code values reside in the heap and take the form $\Lambda\, \Upsilon, [\Delta].IS$ where the logical variables in Υ and the type variables in Δ are bound in the instruction sequence IS.

Word values may also have code type: *extended labels* contain a reference to a heap object, along with an *argument sequence* containing a list of logical terms and types which have been applied to the label. We discuss argument sequences and extended labels further in Sect. 5.1.

5. Operational Semantics

In this section we present the operational semantics for Singleton, along with a discussion of the various instructions. We shall focus our discussion on those instructions which are Singleton-specific. The remainder, at least operationally, are typical of TALs in general; see [14] for more details.

Definition 5.1 The small-step operational semantics of Singleton are denoted by the following judgement

$$P \longmapsto P'$$

where P and P' are program states. The semantics are given in Fig. 5

5.1 Argument Sequences and Extended Labels

Singleton programs may abstract over both logical terms and machine type. An *argument sequence*

$$\Sigma\ =\ (\tilde{A};\ \tilde{\tau})$$

is used to instantiate any such terms, and is simply a tuple containing a sequence of logical arguments and a sequence of type arguments. Given two argument sequences Σ and Σ', $\Sigma \frown \Sigma'$ is their pairwise concatenation.

Argument sequences appear in a number of places in Singleton: as an argument to the `apply` operation and as arguments to branching instructions In addition, some instructions, namely `beq` and `case`, generate argument sequences containing proof terms.

An *extended label* is a label along with a (possibly empty) argument sequence. Operationally, extended label values collect the arguments to a function; partially applying code values complicates the proof of type erasure, and so the actual substitution occurs only when the final value is required, that is, when the label is used as a branch target.

[2] Hiding a proof term is essentially hiding the *existence* of a proof, as CiC has the proof-irrelevance property.

$(H, R, IS_0) \longmapsto P'$

if IS_0 is	and P' is
`move` $r_d,\ r_s;\ IS$	$(H,\ R\{r_d \mapsto R\ r_s\},\ IS)$
`add` $r_d,\ r_1,\ r_2;\ IS$	$(H,\ R\{r_d \mapsto R\ r_1\ +\ R\ r_2\},\ IS)$
`ldi` $r_d,\ n;\ IS$	$(H,\ R\{r_d \mapsto n\},\ IS)$
`pack` $r_d,\ r_s$ `as` τ `hiding` $o;\ IS$	$(H,\ R\{r_d \mapsto pack\ o\ in\ R\ r_s\},\ IS)$
`unpack` $r_d,\ r_s$ `as` $x;\ IS$	$(H,\ R\{r_d \mapsto wv\},\ IS[x\ :=\ v])$ when $R\ r_s\ =\ pack\ v\ in\ wv$
`alloc` $r,\ \tilde{\tau}; IS$	$(H\{l \mapsto \langle ?\tilde{\tau}_0,\ \ldots,\ ?\tilde{\tau}_{|\tilde{\tau}|}\rangle\}, R\{r \mapsto l\ @\ ()\}, IS)$ when l fresh in H
`load` $r_d,\ r_s(n);\ IS$	$(H,\ R\{r_d \mapsto \tilde{v}_n\},\ IS)$ when $R\ r_s = l\ @\ \Sigma$ and $H\ l = \langle\tilde{v}\rangle$
`store` $r_s,\ r_d(n); IS$	$(H\{l \mapsto \langle\tilde{v}\{n \mapsto R\ r_s\}\rangle\}, R, IS)$ when $R\ r_d = l\ @\ \Sigma$ and $H\ l = \langle\tilde{v}\rangle$
`apply` $r_d,\ r_s,\ \Sigma';\ IS$	$(H,\ R\{r_d \mapsto l\ @\ \Sigma \frown \Sigma'\},\ IS)$ when $R\ r_s = l\ @\ \Sigma$
`lda` $r_d,\ \mathcal{L};\ IS$	$(H,\ R\{r_d \mapsto \mathcal{L}\},\ IS)$
`beq` $r_1,\ r_2,\ \mathcal{L}$ `as` $p;\ IS$	$(H,\ R,\ IS_{br})$ when $(pf\ :\ R\ r_1 = R\ r_2)$ and $H\ \models\ \mathcal{L}@\ (pf)\ \triangleright\ IS_{br}$ and $(H,\ R,\ IS[p\ :=\ pf])$ when $(pf\ :\ R\ r_1 \neq R\ r_2)$
`br` $\mathcal{L}$	$(H,\ R,\ IS)$ when $H\ \models\ \mathcal{L}\ \triangleright\ IS$
`jump` r	$(H,\ R,\ IS)$ when $H\ \models\ R\ r\ \triangleright\ IS$
`coerce` $r_d,\ r_s$ `using` $p;\ IS$	$(H,\ R\{r_d \mapsto R\ r_s\},\ IS)$
`inject` $r_d,\ r_s$ `as` $sgl(v : T);\ IS$	$(H,\ R\{r_d \mapsto sgl\ (v :\ T)\ in\ R\ r_s\},\ IS)$
`project` $r_d,\ r_s;\ IS$	$(H,\ R\{r_d \mapsto wv\},\ IS)$ when $R\ r_s = sgl\ (v :\ T)\ in\ wv$
`case` $r_d,\ r_s,\ \tilde{\mathcal{L}}$	$(H, R\{r_d \mapsto l\ @\ \Sigma\}, IS)$ when $R\ r_s = sgl\ (v :\ T)\ in\ (l\ @\ \Sigma)$, $H\ l = \langle n, \ldots\rangle$, $H\ \models\ \mathcal{L}_n@\ (B_1, \ldots, B_m, refl\_equal\ T\ v)\ \triangleright\ IS$

Figure 5. Operational semantics for Singleton.

Substitution of logical terms and machine types into the various syntactic forms is defined in the usual fashion; we thus omit the definition from this paper. As a shorthand we use

$$v[\tilde{x}\ :=\ \tilde{A}][\tilde{\alpha}\ :=\ \tilde{\tau}]$$

for the *sequential* substitution for $\tilde{x}$ and $\tilde{\alpha}$ into v. Furthermore,

$$v[\Upsilon\ :=\ \tilde{A}][\Delta\ :=\ \tilde{\tau}]\ \triangleq\ v[\tilde{x}\ :=\ \tilde{A}][\tilde{\alpha}\ :=\ \tilde{\tau}]$$

where $\Upsilon\ =\ (\tilde{x}\ :\ \tilde{B})$ and $\Delta\ =\ \{\tilde{\alpha}\ ::\ \tilde{\kappa}\}$.

Example 5.1 Recall our *head* example from Sect. 2. If we imagine that `head` is applied to the list $[1,\ 2,\ 3]$, that is, the logical argument a will be nat while xs will be the above list, then the `case` instruction in `head` is passed two extended labels:

$$\texttt{hNil}\ @\ (nat,\ [1,\ 2,\ 3])$$

and

$$\texttt{hCons}\ @\ (nat,\ [1,\ 2,\ 3])$$

Execution of the `case` instruction will find that the list is not empty, and hence will invoke `hCons` with the additional arguments 1, $[2,\ 3]$, and the proof

$$refl\_equal\ (list\ nat)\ [1,\ 2,\ 3]\ :\ [1,\ 2,\ 3]\ =\ [1,\ 2,\ 3]$$

Because `case` will branch to `hCons`, the substitution will be performed, resulting in the instruction sequence

```
load t1, t1(1)
apply ra, ra, (1, [2, 3], refl_equal (list nat) [1, 2, 3])
jump ra
```

where x, xs', and pf have been substituted accordingly.

Definition 5.2 (Application) The judgement

$$\models\ v\ @\ \Sigma\ \triangleright\ IS$$

holds when substituting the arguments in Σ into the code value v results in the instruction sequence IS.

$$\frac{}{\models\ (\Lambda\Upsilon, [\Delta].IS)@(\vec{A}; \vec{\tau})\ \triangleright\ (IS[\Upsilon\ :=\ \vec{A}][\Delta\ :=\ \vec{\tau}])}$$

We extend this to application at an extended label at a heap: given a heap H, an extended label $\mathcal{L}\ =\ l\ @\ \Sigma$, and an argument sequence Σ', we have

$$\frac{Hl = v \qquad \models v@(\Sigma \frown \Sigma')\ \triangleright\ IS}{H\ \models\ \mathcal{L}@\Sigma'\ \triangleright\ IS}$$

Application is rather straightforward: we simply substitute any arguments for the corresponding variables. Although substitution may result in a malformed instruction sequence, that is, one containing ill-typed CiC terms, the corresponding static judgement (Defn. 6.2) ensures that substitution occurs only when it results in well-formed instruction sequences.

We finish this section by noting that application has *no* run-time penalty for a Singleton program. Although substitution seems to create a copy of an instruction sequence, all types are erased in the translation from Singleton into machine code and hence all copies of an instruction sequence are identical after erasure.

5.2 Singleton operations

The singleton operations are perhaps the most novel part of Singleton; in this section we discuss their semantics. Singleton operations manipulate values of the form $sgl\ (v :\ T)\ in\ wv$, where wv is the object representing the singleton. We note, however, that only the `case` operation examines this value, requiring that it be a label referring to a tuple with at least one member, and this member being a word.

Rewriting the associated value. The `coerce` instruction rewrites the value associated with the singleton type using a proof of equality; operationally it is equivalent to a `move` operation.

Injection and projection. The `inject` operation creates singleton values by injecting objects of the representation type; the `project` operation does the converse, projecting the singleton type into the representation type. Operationally, `inject` and `project` simply create and destroy singleton values, respectively.

Case elimination. The `case` operation eliminates singleton values. This instruction, given a singleton object, branches depending on the value associated with the singleton. In addition, `case` extracts the representation object from the singleton and passes the destination label this object, and any constructor arguments and a proof relating the branch taken with the associated value.

Example 5.2 Recall the `case` instruction from `head`.

$$\texttt{case}\ \texttt{t}_1,\ \texttt{a}_1,\ [\texttt{hNil} \mathbin{@} (a,\ xs),\ \texttt{hCons} \mathbin{@} (a,\ xs)]$$

If the list associated with the type of $\texttt{a}_1$ is nil, then `case` will branch to `hNil`, otherwise `case` will branch to `hCons`. In the former case, the only logical argument generated is the proof

$$pf\ :\ xs\ =\ nil$$

while `hCons` gets the arguments from the $cons$ constructor, namely x and xs', along with the proof

$$pf\ :\ xs\ =\ cons\ x\ xs'$$

If we are operating over a singleton with associated type T, then each constructor for T has a corresponding label in the arguments to `case`, where the order of the labels corresponds to the order of the constructors. In general, given the singleton value

$$sgl\ (v:\ T)\ in\ wv$$

where

$$v\ \equiv_{\beta\iota}\ Ctor(n,\ A)\ B_1\ \ldots\ B_m$$

then `case` will branch to the n-th label. We note that, at run-time, all values of an inductive type are equivalent to some fully applied constructor.

The n-th label should refer to a code block with arguments $B_1\ \ldots\ B_m$ along with a proof that

$$v\ =\ Ctor(n,\ A)\ B_1\ \ldots\ B_m$$

Dynamically, this proof is simply the reflexivity of equality at the value v, recalling that equivalent terms in CiC are indistinguishable. The main utility of this proof is to exploit *statically*, in the code for this label, that we are in the correct branch, and hence have discovered the shape of v.

Operationally, this branch target is determined by the representation object wv. A singleton value eliminated by the `case` instruction must have

$$wv\ =\ l \mathbin{@} \Sigma$$

for some label l and argument sequence Σ. Furthermore, this label should point to some tuple in the heap with the tag n; the well-formedness conditions on singletons ensure that this is the case, including that this word is equal to the constructor index.

The new instruction sequence IS is obtained by applying the extended label to $B_1\ \ldots\ B_m$, followed by

$$refl\_equal\ T\ v\ :\ v\ =\ v$$

Finally, the value wrapped by the singleton value constructor is extracted and moved into the destination register.

5.2.1 Existential operations

The existential instructions manipulate values of the form $pack\ v\ in\ wv$. The `pack` operation constructs these values, while the `unpack` operation extracts the hidden v, substituting into the remaining instructions.

Example 5.3 We can implement addition on the non-dependent words from Eg. 4.1 (adding $\texttt{t}_1$ and $\texttt{t}_2$ with the result in $\texttt{t}_1$) by

$$\begin{array}{l}\texttt{unpack}\ \texttt{t}_1,\ \texttt{t}_1\ \texttt{as}\ n_1\\ \texttt{unpack}\ \texttt{t}_2,\ \texttt{t}_2\ \texttt{as}\ n_2\\ \texttt{add}\ \texttt{t}_1,\ \texttt{t}_1,\ \texttt{t}_2\\ \texttt{pack}\ \texttt{t}_1,\ \texttt{t}_1\ \texttt{as}\ WORD\ \texttt{hiding}\ n_1\ +\ n_2\end{array}$$

5.2.2 Code operations

The code operations manipulate code values in the heap, along with labels referring to such values. Only the branching instructions manipulate code variables directly; the instructions `apply` and `lda` manipulate extended labels only. In our semantics application is deferred until the value is actually used, that is, when the label is used as a branch target; thus simplifies the proof of type erasure.

The `apply` operation applies the given logical arguments to the code value in the source register, simply appending the arguments to any existing arguments for that label, while the `lda` operation creates a new value with the given label and arguments.

The branching operations, `beq`, `br`, and `jump`, perform the actual application by substituting the arguments for the corresponding variables. The `beq` instruction also substitues a proof of the equality (or inequality) of the values in the given registers. In all cases, the result of these substitutions forms the new instruction sequence.

6. Static Semantics

The judgements forming Singleton's static semantics are given in Fig. 6. All judgements are modulo CiC equivalence [11]. Again, we shall concentrate primarily on those which are Singleton-specific.

6.1 Well-formed types

A type is well-formed if all logical terms are well-typed and all type variables are accounted for. In addition, we insist that the value associated with a singleton type has sort Set. This allows us to ignore propositions when constructing the representation type.

We classify types into two kinds, *boxed* ($\mathcal{B}$) and *unboxed* ($\mathcal{U}$). This classification is required primarily to ensure Singleton programs can be safely garbage collected, although we will not address garbage collection in this paper.

In essence, a type is boxed if the object it classifies resides on the heap: tuples, singleton types, and code types are boxed, while words are unboxed. Existential types inherit the kind of the type under the existential.

Definition 6.1 (Well-formed types) The judgement

$$\Upsilon; \Delta\ \vdash\ \tau\ ::\ \kappa$$

holds when type τ is well-formed with kind κ under the logical context Υ and type environment Δ.

6.2 Well-formed applications and labels

As noted in Sect. 5.1, at a number of points within a Singleton program we may apply both logical and type arguments to code values. The resulting object is well-formed only when the arguments are well-formed and match those expected by the target.

Definition 6.2 (Well-formed application) The judgement

$$\Upsilon;\ \Delta\ \vdash\ \tau \mathbin{@} \Sigma\ \rhd\ \sigma$$

holds whenever the application of the arguments in Σ to the type τ is valid and results in the type σ, under the logical context Υ and type context Δ.
We extend this to extended labels and heaps.

Constructing representation types $\boxed{\Upsilon \vdash \vec{v} \Downarrow \vec{\tau} \qquad \Upsilon \vdash v \downarrow \tau}$

$$\frac{}{\Upsilon \vdash \epsilon \Downarrow \epsilon} \qquad \frac{\Upsilon \Vdash v : T : \mathit{Set} \qquad \Upsilon \vdash \vec{A} \Downarrow \vec{\tau}}{\Upsilon \vdash v, \vec{A} \Downarrow \mathit{sgl}(v : T), \vec{\tau}} \qquad \frac{\Upsilon \Vdash v : T : s \qquad \Upsilon \vdash \vec{A} \Downarrow \vec{\tau}}{\Upsilon \vdash v, \vec{A} \Downarrow \vec{\tau}} (s \equiv_{\beta\iota} \mathit{Prop}, \mathit{Type})$$

$$\frac{\Upsilon \Vdash n : \mathit{nat}}{\Upsilon \vdash n \downarrow \langle \mathit{word}(n) \rangle} \qquad \frac{\Upsilon \Vdash v : T \qquad \Upsilon \vdash \vec{B} \Downarrow \vec{\tau}}{\Upsilon \vdash v \downarrow \langle \mathit{word}([\![n]\!]), \vec{\tau} \rangle} (T \not\equiv_{\beta\iota} \mathit{nat}, v \equiv_{\beta\iota} \mathit{Ctor}(n, T) B_1 \dots B_m)$$

Well-formed types and contexts $\boxed{\Upsilon; \Delta \vdash \tau :: \kappa \qquad \Upsilon; \Delta \vdash \Gamma \qquad \Upsilon; \Delta \vdash \Psi}$

$$\frac{\alpha :: \kappa \in \Delta}{\Upsilon; \Delta \vdash \alpha :: \kappa} \qquad \frac{\Upsilon \Vdash n : \mathit{nat}}{\Upsilon; \Delta \vdash \mathit{word}(n) :: \mathcal{U}} \qquad \frac{\Upsilon; \Delta \vdash \tau_i :: \kappa_i}{\Upsilon; \Delta \vdash \langle \tau_1^{\phi_1}, \dots, \tau_n^{\phi_n} \rangle :: \mathcal{B}} (i \leq n) \qquad \frac{\Upsilon \Vdash v : T \qquad \Upsilon \Vdash T : \mathit{Set}}{\Upsilon; \Delta \vdash \mathit{sgl}(v : T) :: \mathcal{B}}$$

$$\frac{\Upsilon \Vdash \Phi \qquad \Upsilon, \Phi; \Delta, \Omega \vdash \Gamma}{\Upsilon; \Delta \vdash \forall \Phi, [\Omega].\Gamma :: \mathcal{B}}$$

$$\frac{\Gamma\, r = \tau \qquad \Upsilon; \Delta \vdash \tau :: \kappa}{\Upsilon; \Delta \vdash \Gamma} (r \in \mathit{dom}(\Gamma)) \qquad \frac{\Psi\, l = \tau \qquad \Upsilon; \Delta \vdash \tau :: \mathcal{B}}{\Upsilon; \Delta \vdash \Psi} (l \in \mathit{dom}(\Psi))$$

Well-formed application $\boxed{\Upsilon; \Delta \vdash \tau @ (\vec{A}; \vec{\sigma}) \rhd \xi}$

$$\frac{}{\Upsilon; \Delta \vdash \tau @ () \rhd \tau} \qquad \frac{\Upsilon; \Delta \vdash \sigma :: \kappa \qquad \Upsilon; \Delta \vdash \forall \Phi, [\Omega].\Gamma[\alpha := \sigma] @ (\vec{\tau}) \rhd \xi}{\Upsilon; \Delta \vdash \forall \Phi, [\alpha :: \kappa, \Omega].\Gamma @ (\sigma, \vec{\tau}) \rhd \xi}$$

$$\frac{\Upsilon \Vdash b : A \qquad \Upsilon; \Delta \vdash (\forall \Phi, [\Omega].\Gamma)[x := b] @ (\vec{B}; \vec{\tau}) \rhd \xi}{\Upsilon; \Delta \vdash \forall (x : A) \Phi, [\Omega].\Gamma @ (b, \vec{B}; \vec{\tau}) \rhd \xi}$$

Initialisation subtype and well-formed values $\boxed{\vdash \sigma \leq \tau \qquad \Upsilon; \Delta; \Psi \vdash wv : \tau \qquad \Psi \vdash hv : \tau}$

$$\frac{}{\vdash \tau \leq \tau} \qquad \frac{\vdash \sigma \leq \omega \qquad \vdash \omega \leq \tau}{\vdash \sigma \leq \tau} \qquad \frac{}{\vdash \langle \dots, \tau^0, \dots \rangle \leq \langle \dots, \tau^1, \dots \rangle}$$

$$\frac{}{\Upsilon; \Delta; \Psi \vdash n : \mathit{word}([\![n]\!])} \qquad \frac{\Upsilon; \Delta; \Psi \vdash \mathcal{L} \rhd \sigma \qquad \vdash \tau \leq \sigma}{\Upsilon; \Delta; \Psi \vdash \mathcal{L} : \tau} \qquad \frac{\Upsilon \Vdash v : A \qquad \Upsilon; \Delta; \Psi \vdash wv : \tau[x := v]}{\Upsilon; \Delta; \Psi \vdash \mathit{pack}\ v\ \mathit{in}\ wv : \exists (x : A).\tau}$$

$$\frac{\Upsilon \Vdash v : T \qquad \Upsilon \vdash v \downarrow \tau \qquad \Upsilon; \Delta; \Psi \vdash wv : \tau}{\Upsilon; \Delta; \Psi \vdash \mathit{sgl}\ (v : T)\ \mathit{in}\ wv : \mathit{sgl}(v : T)}$$

$$\frac{\epsilon; \epsilon; \Psi \vdash wv_i : \tau_i}{\Psi \vdash \langle \vec{wv} \rangle : \langle \vec{\tau} \rangle} (i \leq |\vec{wv}| \text{ and } \tau_i \text{ initialised}) \qquad \frac{\Upsilon; \Delta; \Psi; \Gamma \vdash \mathit{IS}}{\Psi \vdash \Lambda \Upsilon, [\Delta].\mathit{IS} : \forall \Phi, [\Delta].\Gamma}$$

Instruction sequence prefixes and heap membership $\boxed{\mathit{IS} \sqsubseteq \mathit{IS}' \qquad \vdash \mathit{IS} \in H}$

$$\frac{}{\mathit{IS} \sqsubseteq \mathit{IS}} \qquad \frac{\mathit{IS} \sqsubseteq \mathit{IS}'[x := p]}{\mathit{IS} \sqsubseteq \texttt{beq}\ r_1, r_2, \mathcal{L}\ \texttt{as}\ x; \mathit{IS}'} \qquad \frac{\mathit{IS} \sqsubseteq \mathit{IS}'[x := v]}{\mathit{IS} \sqsubseteq \texttt{unpack}\ r_d,\ r_s\ \texttt{as}\ x; \mathit{IS}'} \qquad \frac{\mathit{IS} \sqsubseteq \mathit{IS}'}{\mathit{IS} \sqsubseteq I; \mathit{IS}'} (I \text{ not } \texttt{beq}, \texttt{unpack})$$

$$\frac{H l = \Lambda \Upsilon, [\Delta].\mathit{IS}' \qquad \mathit{IS} \sqsubseteq \mathit{IS}'[\Upsilon := \vec{A}][\Delta := \vec{\tau}]}{\vdash \mathit{IS} \in H}$$

Well-formed heaps, register files, and programs $\boxed{\vdash H : \Psi \qquad \Upsilon; \Delta; \Psi \vdash R : \Gamma \qquad \vdash P}$

$$\frac{\Psi\, l = \tau \qquad H\, l = hv \qquad \Psi \vdash hv : \tau}{\vdash H : \Psi} (l \in \mathit{dom}(\Psi)) \qquad \frac{\Gamma\, r = \tau \qquad R\, r = wv \qquad \Upsilon; \Delta; \Psi \vdash wv : \tau}{\Upsilon; \Delta; \Psi \vdash R : \Gamma} (r \in \mathit{dom}(\Gamma))$$

$$\frac{\vdash H : \Psi \qquad \epsilon; \epsilon; \Psi \vdash R : \Gamma \qquad \epsilon; \epsilon; \Psi; \Gamma \vdash \mathit{IS} \qquad \vdash \mathit{IS} \in H}{\vdash (H, R, \mathit{IS})}$$

Figure 6. Singleton static semantics judgements (not including instruction sequences). We construct CiC natural numbers from meta-logical numbers using $[\![n]\!]$.

$$\frac{\Psi\ l = \tau \qquad \Upsilon; \Delta \vdash \tau @ (\Sigma_1 \frown \Sigma_2) \rhd \sigma}{\Upsilon; \Delta; \Psi \vdash (l @ \Sigma_1) @ \Sigma_2 \rhd \sigma}$$

Applications are only relevant for code types, although we allow the empty argument sequence to be applied to any type. Otherwise, we apply the arguments from left to right, substituting into the code type for the corresponding variable. An application is well-formed only if the logical arguments are of the expected logical type, and the type arguments are of the expected kind.

Example 6.1 Consider again Eg. 5.1, where we apply `head` to the logical arguments nat and $[1,\ 2,\ 3]$. Recall the type of `head`

$$\begin{array}{l}\forall(a\ :\ Set)(xs\ :\ list\ a).\\ \ \{\mathtt{a}_1\ :\ sgl(xs : list\ a),\\ \ \ \mathtt{ra}\ :\ \forall(x\ :\ list\ a)(xs'\ :\ list\ a)\\ \qquad (pf\ :\ xs\ =\ cons\ x\ xs').\{\mathtt{t}_1\ :\ sgl(x : a)\}\}\end{array}$$

We shall call this type $\tau_{\mathtt{head}}$ in the following.
To apply nat, we must show

$$\Upsilon \Vdash\ nat\ :\ Set$$

noting that the first argument, a, has type Set. This holds, and so we substitute nat for a to get

$$\begin{array}{l}\forall(xs\ :\ list\ nat).\\ \ \{\mathtt{a}_1\ :\ sgl(xs : list\ nat),\\ \ \ \mathtt{ra}\ :\ \forall(x\ :\ list\ nat)(xs'\ :\ list\ nat)\\ \qquad (pf\ :\ xs\ =\ cons\ x\ xs').\{\mathtt{t}_1\ :\ sgl(x : nat)\}\}\end{array}$$

To apply $[1,\ 2,\ 3]$, we must show

$$\Upsilon \Vdash\ [1,\ 2,\ 3]\ :\ list\ nat$$

as xs has type $list\ nat$. Again this holds, and so, letting $\sigma_{\mathtt{head}}$ be

$$\begin{array}{l}\{\mathtt{a}_1\ :\ sgl([1,\ 2,\ 3] : list\ nat),\\ \ \mathtt{ra}\ :\ \forall(x\ :\ list\ nat)(xs'\ :\ list\ nat)\\ \qquad (pf\ :\ [1,\ 2,\ 3]\ =\ cons\ x\ xs').\{\mathtt{t}_1\ :\ sgl(x : nat)\}\}\end{array}$$

we can derive $\Upsilon;\ \Delta\ \vdash\ \tau_{\mathtt{head}}\ @\ (nat,\ [1,\ 2,\ 3])\ \rhd\ \sigma_{\mathtt{head}}$.

6.3 Pseudo-elimination judgements

In this section we define auxiliary judgements used to eliminate singleton values. A *representation type* is the machine type underlying the singleton type, and the *elimination candidate* describes the possible forms the value associated with the singleton can take.

6.3.1 Representation types

We have informally introduced the concept of a representation type; in this section we define it formally.

Definition 6.3 (Representation types) The judgement

$$\Upsilon\ \vdash\ v\ \downarrow\ \tau$$

holds when the value v is represented by the type τ under the logical context Υ.

In general, the representation type for a given value is a tuple containing the constructor index for that value as a word, followed by singleton types for those constructor arguments with informative types (that is, types in the sort Set). Natural numbers, however, are represented using a word type boxed by a tuple.

Example 6.2 We can construct the representation type for the list of natural numbers $[1,\ 2,\ 3]$, as

$$\begin{array}{l}\Upsilon\ \vdash\ [1,\ 2,\ 3]\ \downarrow\ \langle word(1),\ sgl(1 : nat),\\ \qquad\qquad sgl([2,\ 3] : list\ nat)\rangle\end{array}$$

which matches our description from Sect. 2.1.2.

Example 6.3 The type of lists containing elements which satisfy some predicate P is

$$\begin{array}{l}\textbf{Inductive}\ listP\ (T\ :\ Set)(P\ :\ T\ \rightarrow\ Prop)\ :\ Set\ :=\\ \ \mid nilP\ :\ listP\ T\ P\\ \ \mid consP\ :\ \forall(v\ :\ T),\ P\ v\ \rightarrow\ listP\ T\ P\ \rightarrow\ listP\ T\ P\end{array}$$

The list corresponding to $[1,\ 2,\ 3]$ then has the representation type

$$\begin{array}{l}\Upsilon\ \vdash\ [1,\ 2,\ 3]\ \downarrow\ \langle word(1),\ sgl(1 : nat),\\ \qquad\qquad sgl([2,\ 3] : listP\ nat\ P)\rangle\end{array}$$

The proofs terms, having sort $Prop$, do not appear as elements in the representation type.

6.3.2 Elimination Candidates

Recall from Sect. 2.1.2 that the type of a label argument to `case` depends upon the corresponding constructor; the types of `hNil` and `hCons`, for example, are partially determined by their being the targets for the nil and $cons$ cases, respectively. In this section we define the *elimination candidates* for a given .

Definition 6.4 (Elimination candidates) Given a value, v, of inductive type T, and n, where

$$\begin{array}{l}\textbf{Inductive}\ T\ :\ Set\ :=\\ \ \vdots\\ \ \mid C_n : \forall(x_1\ :\ A_1)\ \ldots\ (x_m\ :\ A_m),\ T\\ \ \vdots\end{array}$$

the elimination candidate for v, T, and n is

$$\begin{array}{l}elims\ v\ T\ n\ =\ (Ctor(n,\ T)\ x_1\ \ldots\ x_m,\\ \qquad\qquad (x_1\ :\ A_1)\ \ldots\ (x_m\ :\ A_m)\\ \qquad\qquad (p\ :\ v\ =\ Ctor(n,\ T)\ x_1\ \ldots\ x_m))\end{array}$$

where the x_i and p are chosen to be fresh.

The elimination candidate represents the possible head-normal forms for the value v, with new variables for the (unknown) constructor arguments. The candidate consists of the head-normal form, which is used to construct the representation type, and a context containing these new variables along with a proof that v equals the head-normal form, used to construct the target label type.

Example 6.4 The elimination candidates for $list\ T$ and some value v are

$$\begin{array}{l}elims\ v\ (list\ T)\ 0 =\ (nil,\ \epsilon)\\ elims\ v\ (list\ T)\ 1 =\ (cons\ a\ as, (a\ :\ T)(as\ :\ list\ T)\\ \qquad\qquad\qquad\qquad\qquad (p\ :\ v\ =\ cons\ a\ as))\end{array}$$

where a, as, and p are fresh. The context part of each candidate gives the extra arguments for the `hNil` and `hCons` labels.

Example 6.5 The elimination candidates for $listP\ T\ P$ and some value v are

$$\begin{array}{l}elims\ v\ (listP\ T\ P)\ 0 =\ (nilP,\ \epsilon)\\ elims\ v\ (listP\ T\ P)\ 1 =\ (consP\ a\ p\ as,\\ \qquad\qquad\qquad\qquad (a\ :\ T)(p\ :\ P\ a)\\ \qquad\qquad\qquad\qquad (as\ :\ listP\ T\ P)\\ \qquad\qquad\qquad\qquad (q\ :\ v\ =\ consP\ a\ p\ as))\end{array}$$

where a, as, p, and q are fresh. Note that, unlike the representation type for $listP$ (see Eg. 6.3), the elimination candidates *are* effected by the extra proof term: the environment portion of $elims\ v\ (listP\ T\ P)\ 1$ includes the proof p.

These operations, and hence singleton types, are defined only for inductive types without arguments[3]: constructing useful elimination candidates for inductive families is rather involved. Supporting them is left as future work.

6.4 Well-formed instruction sequences

The judgement for well-formed instruction sequences forms the heart of the type system for Singleton.

Definition 6.5 (Well-formed instruction sequence) The judgement

$$\Upsilon;\ \Delta;\ \Psi;\ \Gamma\ \vdash\ IS$$

holds when the instruction sequence IS is well-formed under the logical context Υ, type context Δ, heap type Ψ, and register file type Γ. The rules for this judgement are given in Fig. 7.

Arithmetic operations. The typing rules for arithmetic operations are more specific than with a traditional TAL as they must track the operation performed in the type of the target register: in the addition case, for example, we have that the destination register has type $word(n_1 + n_2)$ where n_1 and n_2 are the word values corresponding to the two argument registers. The rule for loading an immediate also produces a word type.

Existential operations. The typing rules for existential types are straightforward: the `pack` instruction hides a logical term value in a type, so we must check that when we substitute the given value back into the type we get that of the source register. Furthermore, we must check that value being hidden is well-formed.

The `unpack` operation performs the opposite operation, obtaining previously hidden values. We abuse α-conversion here so that the names bound by the existential are identical to the those of the target variables. Thus, we simply extend the logical and type contexts and update the target register with the body of the existential.

Singleton operations. The typing rules relating to singleton operations rely on the auxiliary judgements from Sect. 6.3.

The `inject` operation constructs a singleton type, and thus requires that the type of the source register is the representation type for the target singleton type.

Conversely, the `project` operation destroys a singleton with a specific value, and thus simply updates the destination register with the representation type at that value. Note that this operation is well-typed only when the representation type exists; in the case of non-primitive singleton types, this means that the head of the associated value is equivalent to a constructor.

The `coerce` operation rewrites the singleton value using a proof of equality; we must check that the proof is actually an equality, and that the source register is a singleton with a value equivalent to the left hand side of the equality. This operation is also defined for word types; this case is similar to the general singleton case.

The `case` operation eliminates a non-primitive singleton type by case analysis: the elimination candidates for this type give all possibilities for the associated value, introducing new variables for unknown constructor arguments (see Defn. 6.4). For each elimination candidate the corresponding label must refer to a code type which abstracts over the elimination context. Furthermore, the target register file type must be a subset of the current register context after updating the destination register with the representation type for the elimination value.

Code operations. The branching operations require that the target is fully applied: the `br` operation is well formed if the given (extended) label points to a fully-applied code type, the `jump` operation is similar except that the target label is sourced from a register. The `beq` instruction is slightly different in that the target type abstracts over an equality proof for the argument words. As usual, all instructions require the target register file to be a subset of the current register file.

The `apply` operation is well formed if the arguments are applicable at the type of the source register; application gives the new type, which is used to update the destination register.

Similarly, loading a constant label differs from a traditional TAL only in that the immediate is an extended label. The rule for the `lda` operation then updates the target register with the heap type at the given label, taking into account any arguments.

6.5 Well-formed values and programs

The well-formed value judgements are used to show that register files and heaps are well-formed, and hence that programs are well-formed. These judgements are standard, taking into consideration any extended labels.

Following Hamid et al. [6], a well-formed program requires that the instruction sequence resides in the heap. This property, required to show type erasure, is rather more complicated in Singleton than in a traditional TAL.

Definition 6.6 (Heap suffix membership) The judgement

$$\vdash\ IS\ \in\ H$$

holds when IS is a suffix of some object in the heap H.

In essence, this judgement states that IS can be obtained from some code value in the heap H by dropping instructions and instantiating any bound variables.

Definition 6.7 (Well-formed program) The judgement

$$\vdash\ P$$

holds when P is a well-formed program.

This holds when for some heap type Ψ and register file type Γ, the heap, register file, and instruction sequence are all well-formed, and the instruction sequence exists in the heap.

7. Type Safety and Type Erasure

We have developed [14] a machine-checked model of Singleton in the system Coq, showing both type safety and type erasure properties. Space constraints require that we only broadly discuss the proofs of these properties; the Coq scripts for these proofs are available at

`http://www.cse.unsw.edu.au/~sjw/thesis/proofs/`

An important lemma in the type-safety proof is the soundness of the well-formed application judgement; we show the following progress- and preservation-like lemmas for applications.

Lemma 1 If $\Psi \vdash hv : \tau$ and $\Upsilon; \Delta \vdash \tau@\Sigma \rhd \sigma$ then $\models hv@\Sigma \rhd hv'$ for some hv'.

We note that all CiC terms in hv' are well-formed under the context Υ.

Lemma 2 If $\Psi \vdash hv : \tau$, and $\Upsilon; \Delta \vdash \tau@\Sigma \rhd \Gamma$, and $\models hv@\Sigma \rhd IS$ then $\Psi \vdash IS : \Gamma$.

We show type safety using the usual progress and preservation lemmas.

Lemma 3 (Progress) If $\vdash (H, R, I)$ then $I = \texttt{halt}\ [\tau]$ or $(H, R, I) \longmapsto P$ for some P.

[3] Parameters, that is, arguments to the type which are constant for a given type are defined; this is the case for T and P for $listP\ T\ P$ above.

$\boxed{\Upsilon; \Delta; \Psi; \Gamma \vdash IS}$

$$\frac{\Gamma\ r_1 = word(n_1) \qquad \Gamma\ r_2 = word(n_2) \qquad \Upsilon; \Delta; \Psi; \Gamma\{r_d \mapsto word(n_1 + n_2)\} \vdash IS}{\Upsilon; \Delta; \Psi; \Gamma \vdash \texttt{add}\ r_d,\ r_1,\ r_2; IS} \qquad \frac{\Upsilon; \Delta; \Psi; \Gamma\{r_d \mapsto \Gamma\ r_s\} \vdash IS}{\Upsilon; \Delta; \Psi; \Gamma \vdash \texttt{move}\ r_d,\ r_s; IS} \qquad \frac{\Upsilon; \Delta; \Psi; \Gamma\{r \mapsto word([\![n]\!])\} \vdash IS}{\Upsilon; \Delta; \Psi; \Gamma \vdash \texttt{ldi}\ r,\ n; IS}$$

$$\frac{\Gamma\ r_s = \langle \ldots, \tau_n^{\mathbf{1}}, \ldots \rangle \qquad \Upsilon; \Delta; \Psi; \Gamma\{r_d \mapsto \tau_n\} \vdash IS}{\Upsilon; \Delta; \Psi; \Gamma \vdash \texttt{load}\ r_d,\ r_s(n); IS} \qquad \frac{\Gamma\ r_s = \tau_n \qquad \Gamma\ r_d = \langle \ldots, \tau_n^{\varphi}, \ldots \rangle \qquad \Upsilon; \Delta; \Psi; \Gamma\{r_d \mapsto \langle \ldots, \tau_n^{\mathbf{1}}, \ldots \rangle\} \vdash IS}{\Upsilon; \Delta; \Psi; \Gamma \vdash \texttt{store}\ r_s,\ r_d(n); IS}$$

$$\frac{\Upsilon; \Delta \vdash \tau_i :: \kappa_i \qquad \Upsilon; \Delta; \Psi; \Gamma\{r \mapsto \langle \vec{\tau}^{\mathbf{0}} \rangle\} \vdash IS}{\Upsilon; \Delta; \Psi; \Gamma \vdash \texttt{alloc}\ r,\ \vec{\tau}; IS}(i \leq |\vec{\tau}|)$$

$$\frac{\tau = \exists(x : T).\xi \qquad \Upsilon \Vdash v : T \qquad \Upsilon; \Delta \vdash \tau :: \pi \qquad \Gamma\ r_s = \xi[x := v] \qquad \Upsilon; \Delta; \Psi; \Gamma\{r_d \mapsto \tau\} \vdash IS}{\Upsilon; \Delta; \Psi; \Gamma \vdash \texttt{pack}\ r_d, r_s\ \texttt{as}\ \tau\ \texttt{hiding}\ v; IS} \qquad \frac{\Gamma\ r_s = \exists(x : A).\tau \qquad \Upsilon, x : A; \Delta; \Psi; \Gamma\{r_d \mapsto \tau\} \vdash IS}{\Upsilon; \Delta; \Psi; \Gamma \vdash \texttt{unpack}\ r_d,\ r_s\ \texttt{as}\ x; IS}$$

$$\frac{\Gamma\ r_s = \tau \qquad \Upsilon \vdash v \downarrow \tau \qquad \Upsilon; \Delta; \Psi; \Gamma\{r_d \mapsto sgl(v : T)\} \vdash IS}{\Upsilon; \Delta; \Psi; \Gamma \vdash \texttt{inject}\ r_d,\ r_s\ \texttt{as}\ sgl(v : T); IS} \qquad \frac{\Gamma\ r_s = sgl(v : T) \qquad \Upsilon \vdash v \downarrow \tau \qquad \Upsilon; \Delta; \Psi; \Gamma\{r_d \mapsto \tau\} \vdash IS}{\Upsilon; \Delta; \Psi; \Gamma \vdash \texttt{project}\ r_d,\ r_s; IS}$$

$$\frac{\Gamma\ r_s = sgl(v : T) \qquad \Upsilon \Vdash p : v = u \qquad \Upsilon; \Delta; \Psi; \Gamma\{r_d \mapsto sgl(u : T)\} \vdash IS}{\Upsilon; \Delta; \Psi; \Gamma \vdash \texttt{coerce}\ r_d,\ r_s\ \texttt{using}\ p; IS}$$

$$\frac{\Gamma\ r_s = word(n) \qquad \Upsilon \Vdash p : n = m \qquad \Upsilon; \Delta; \Psi; \Gamma\{r_d \mapsto word(m)\} \vdash IS}{\Upsilon; \Delta; \Psi; \Gamma \vdash \texttt{coerce}\ r_d,\ r_s\ \texttt{using}\ p; IS}$$

$$\frac{\begin{array}{c}\Gamma\ r_s = sgl(v : T) \\ \Upsilon; \Delta; \Psi \vdash \mathcal{L}_i \rhd \forall \Phi_i.\Theta_i \qquad \Upsilon \vdash w_i \downarrow \tau_i \qquad \Theta_i \subseteq \Gamma\{r_d \mapsto \tau_i\} \\ \text{for } i \text{ s.t. } \mathit{elims}\ v\ T\ i = (w_i, \Phi_i)\end{array}}{\Upsilon; \Delta; \Psi; \Gamma \vdash \texttt{case}\ r_d,\ r_s,\ \vec{\mathcal{L}}}(T \not\equiv_{\beta\iota} nat)$$

$$\frac{\Gamma\ r_1 = word(n_1) \qquad \Gamma\ r_2 = word(n_2) \qquad \Upsilon; \Delta; \Psi \vdash \mathcal{L} \rhd \forall(p : n_1 = n_2).\Theta \qquad \Theta \subseteq \Gamma \qquad \Upsilon, p : n_1 \neq n_2; \Delta; \Psi; \Gamma \vdash IS}{\Upsilon; \Delta; \Psi; \Gamma \vdash \texttt{beq}\ r_1, r_2, \mathcal{L}\ \texttt{as}\ p; IS} \qquad \frac{\Upsilon; \Delta; \Psi \vdash \mathcal{L} \rhd \Theta \qquad \Theta \subseteq \Gamma}{\Upsilon; \Delta; \Psi; \Gamma \vdash \texttt{br}\ \mathcal{L}} \qquad \frac{\Gamma\ r = \Theta \qquad \Theta \subseteq \Gamma}{\Upsilon; \Delta; \Psi; \Gamma \vdash \texttt{jump}\ r}$$

$$\frac{\Upsilon; \Delta; \Psi \vdash \mathcal{L} \rhd \tau \qquad \Upsilon; \Delta; \Psi; \Gamma\{r \mapsto \tau\} \vdash IS}{\Upsilon; \Delta; \Psi; \Gamma \vdash \texttt{lda}\ r,\ \mathcal{L}; IS} \qquad \frac{\Upsilon; \Delta \vdash (\Gamma\ r_s)@\Sigma \rhd \tau \qquad \Upsilon; \Delta; \Psi; \Gamma\{r_d \mapsto \tau\} \vdash IS}{\Upsilon; \Delta; \Psi; \Gamma \vdash \texttt{apply}\ r_d,\ r_s,\ \Sigma; IS} \qquad \frac{\Gamma\ \mathtt{r}_0 = \tau}{\Upsilon; \Delta; \Psi; \Gamma \vdash \texttt{halt}\ [\tau]}$$

Figure 7. Well-formed instruction sequences. Recall that $[\![n]\!]$ is the the logical value corresponding to the natural number n.

Lemma 4 (Preservation) If $\vdash P$ and $P \longmapsto P'$ then $\vdash P'$.

Singleton is rather type-heavy, so we show that any well-formed program can be erased to a simple machine state, such that evaluation of the program is matched by machine transitions.

Theorem 5 (Erasure) If $\vdash P$, and $P \longmapsto P'$ and P erases to the machine state s, then s steps (in zero or more steps) to some state s' such that P' erases to s'

We note that the erasure proof is similar to the proofs required to construct a syntactic FPCC [6] system. The main difference lies in the multiple machine steps required by the translation of the `case` instruction: the FPCC approach requires that each machine step correspond to some program step.

The `case` instruction is translated into a computed jump into a table of branches, one entry for each label argument to `case`. The exact steps are (ignoring the register move component): (1) load the index of the tuple under consideration into a temporary register; (2) use this address to calculate the address of a branch instruction inside the array of branches; (3) jump to this computed address; and (4) branch to the target label.

While we have presented the simpler `case` instruction, we believe that it is possible to encode each step as a separate instruction, allowing a straightforward FPCC proof.

8. Related Work

The *type language* (TL) of Shao *et al.* [12] is a variant of the calculus of inductive constructions. TL is intended to be a general-purpose type language for certified programs. As a specific instance, the authors present λ_H, a lambda calculus with singleton words and booleans, conditionals, fixpoints, existentials, and tuples. Existentials hide TL terms, and thus both computational types and logical terms and proofs. The terms in the computational language are outside TL.

TL is used as both the assertion logic and as a language for encoding computational types, which are simply terms in some inductive kind. Thus, it is possible to define multiple classes of computational types; the authors use this to encode their types for λ_H and for a CPS-converted language. Furthermore, types built from existing types need no extra machinery.

We cannot use TL directly to implement Singleton: TL does not include a facility to eliminate inductive kinds as required to construct representation types. We can, however, construct a similar system if we make the representation type explicit in the generalised singleton type.

We use something similar in our encoding of Singleton in the Coq system, described in [14]. The implementation using TL is, due to impredicativity, somewhat simpler; in particular, abstracting over types and representation functions is straightforward.

This approach is not without problems: the representation function can be an arbitrary function, thus making type checking incomplete. The representation function does not need to be a single-level

elimination, and so our practice of checking `case` instructions by considering all possible constructors will not necessarily yield a concrete computational type.

Finally, we note that the sgl constructor is not actually required; we can use $f\ v$ instead of $sgl\ t\ f\ v$, that is, we identify a singleton and its representation type. Unfortunately, these three pieces of information, the type, the function, and the value, are all required by the `case` instruction: we perform case analysis on v, and discovering f and v through unification is, in general, undecidable .

The *logical type theory* (LTT) of Crary and Vanderwaart [4] takes a similar approach to TL, using linear LF [3] as the assertion logic rather than CiC; see [4, Sect. 7] for a comparison. The core computation language of LTT is the higher-order polymorphic lambda calculus extended with computational products and dependent products over proof kinds and families. The language is instantiated by giving a signature defining the syntax of the assertion logic and language primitives.

The considerable difference between CiC and LF makes it hard to compare Singleton in detail to LTT. Assuming a variant of LLT whose computation language is an assembly language, rather than a lambda calculus, the main difference to Singleton types is clearly the lack of inductive types in LTT.

Our treatment of inductive data types is similar to the *guarded recursive datatype constructors* of Xi et al.[16]; in particular, they show how datatype families can be considered as a combination of recursive, sum, and existential types. Although this approach generalises our representation types, it requires the addition of dependent kinds to our type system; whether their approach is feasible in our system we leave to future work.

Our singleton types are similar to refinement types [1, 5, 13]: the type $\{\ x\ :\ T \mid P\ \}$ is a refinement of the type T such that P holds, where x is bound in P. We can simulate such types in Singleton by

$$\begin{array}{l}\textbf{Inductive}\ refinement\ (t\ :\ Set)(P\ :\ t\ \rightarrow\ Prop)\ :\ Set\ :=\\ \quad \mid refinementI\ :\ \forall(v\ :\ t),\ P\ v\ \rightarrow\ refinement\ t\ P\end{array}$$

Although refinement types subsume our singleton types, in that the refinement types $\{x\ :\ T \mid x\ =\ v\}$ corresponds to the type $sgl(v : T)$, we cannot simply replace singleton types by refinement types: our use of representation types requires an associated value. Furthermore, Singleton's use of explicit proof terms obviates the need for refinement types as any restrictions can be more conveniently encoded as a proof assumption.

Xi and Harper[15] propose a *dependently typed assembly language* (DTAL) with singleton word types and length indexed array types. The annotations in DTAL are formulas of linear arithmetic; the typing rules accumulate contexts of refinement types relating to the variables in these formulas. In addition, code blocks can quantify over variables with refinement type; these variables may appear in the indicies of word and array types. As with our system, these indicies limit the mutability of constrained types.

A major difference between DTAL and Singleton lies in the treatment of proofs: in DTAL proofs are discovered by the type checker using linear constraint solvers, compared to the explicit proof-terms found in Singleton. While this reduces the size of type annotations, it requires type indices to be decidable. In addition, constraint satisfaction in DTAL is NP-complete, allowing potential denial-of-service attacks on the type checker. Finally, we note that DTAL programs can be converted into Singleton programs, assuming the constraint satisfaction solver is certifying.

Harren and Necula [7] give a dependently typed assembly language used as a target for their CCured tool [10]. The system is parameterised by a type policy, which takes the form of a set of constraints and constants, along with operations for refining sets of these constraints when given new information. The constants are used to construct expressions for register types.

This system, unlike Singleton, allows mutable dependent fields. This comes at a cost, however, as inter-record dependencies are not allowed: constraints can refer only to the current record. It is unclear as to whether, and to what extent, constraints can refer to records contained in the current record. We note that, like DTAL, type checking requires constraint operations to be decidable.

9. Discussion

While we have given Singleton as an idealised TAL, we believe that it can be extended to support the usual features, such as a stack, of an assembly language.

We presented Singleton as a TAL, although we believe the *idea* of Singleton is not restricted to low-level languages. We chose such a language in order to emphasise the difference between the computational language and the assertion language. However, higher-level languages are certainly also possible.

Acknowledgements The authors wish to thank Gerwin Klein and Toby Murray for their valuable feedback.

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Econ- omy and the Australian Research Council through the ICT Centre of Excellence program.

References

[1] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *Proc. of CSF'08*, pages 17–32. IEEE Computer Society, 2008.

[2] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[3] I. Cervesato and F. Pfenning. A linear logical framework. *Information and Computation*, 179(1):19 – 75, November 2002.

[4] K. Crary and J. C. Vanderwaart. An expressive, scalable type theory for certified code. In *Proc. of ICFP'02*. ACM SIGPLAN, Oct. 2002.

[5] T. Freeman and F. Pfenning. Refinement types for ml. In *Proc. of PLDI'91*, pages 268–277. ACM, 1991.

[6] N. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. *Journal of Automated Reasoning*, 31(3-4):191–229, 2003.

[7] M. Harren and G. C. Necula. Using dependent types to certify the safety of assembly code. In *SAS'05*, pages 155–170, 2005.

[8] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. ISSN 0001-0782.

[9] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *TOPLAS*, 21(3):527–568, 1999.

[10] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proc. of POPL'02*, pages 128–139, 2002.

[11] C. Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In *Proc. of TLCA'93*, pages 328–345. Springer-Verlag, 1993.

[12] Z. Shao, V. Trifonov, B. Saha, and N. Papaspyrou. A type system for certified binaries. *ACM Trans. Program. Lang. Syst.*, 27(1):1–45, 2005.

[13] T. Terauchi. Dependent types from counterexamples. In *Proc. of POPL'10*, pages 119–130, 2010.

[14] S. Winwood. *Singleton: A general-purpose dependently-typed assembly language*. PhD thesis, The University of New South Wales, 2010.

[15] H. Xi and R. Harper. A dependently typed assembly language. *ACM SIGPLAN Notices*, 36(10):169–180, Oct. 2001. ISSN 0362-1340.

[16] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *POPL'03*, pages 224–235, 2003.