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

$$\text{map} : \forall a b. (a \to b) \to \text{list } a \to \text{list } b$$

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

$$(\forall (a : \text{Set})(b : \text{Set})(f : a \to b)(\text{xs} : \text{list } a), \{ a_1 : \forall (v : a). (a_1 : \text{sgl}(v : a)), a_2 : \text{sgl}(\text{xs} : \text{list } a), ra : \{ t_1 : \text{sgl}(\text{map } f \text{ xs} : \text{list } b) \})$$

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 $a_1$ which computes $f v$ for any $v$, a generalised singleton object in $a_2$ 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 $t_1$ 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:
We give two short examples to introduce Singleton and also to serve as a running example through the remainder of this paper.

2. Singleton by Example

We give an operational and a static semantics for Singleton (Sections 3 and 4).

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.

2.1 Taking the head of a list

Consider the list data type

Inductive list (t : Set) : Set :=
| nil : list t
| cons : t → list t → list t

and the head function

head :: ∀a. list a → a
head xs = match xs with
| nil → error
| cons x xs’ → x

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

head :: ∀a. list a → a

and that for head

∀(a : Set)(xs : list a).
\{a₁ : sgl(xs : list a),
ra : ∀(x : list a)(xs’ : list a)
(px : xs = cons x xs’).\{t₁ : sgl(x : a)\}\}
Thus, depending on the contents of $xs$, the register $a_1$ may contain a tuple with a single member, or a tuple with three members. We can then say that the argument in $a_1$ corresponds to the argument to $\text{head}$, $xs$, in that the first element of $xs$ is the second entry in the tuple in $a_1$, 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 $\text{head}$

We now turn our attention to the actual implementation of $\text{head}$. 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 $\text{nil}$ or $\text{cons } x x'$ for some $x$ and $x'$. In fact, this is precisely the behaviour of case: the instruction

$$\text{case } t_1, a_1, [\text{hNil } @ (a, xs), \text{hCons } @ (a, xs)]$$

examines the singleton value in $a_1$ and branches to hNil if the value associated with $a_1$ is $\text{nil}$ and to hCons if the value is $\text{cons}$. The unpacked singleton value is passed in the first element of a representation object and the associated value.

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 $\text{cons}$ constructor, $x$ and $xs'$, along with a proof that these do actually form $xs$, namely

$$pf : xs = \text{cons } x x'$$

Furthermore, the type of register $t_2$ is the representation type for $\text{cons } x x'$, 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

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

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 $\text{head}$

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

$$\text{totalHead} :: \forall (a : \text{Set})(x : \text{list } a)(x' : \text{list } a). (pf : xs = \text{cons } x x'). \\
\{a_1 : \text{sgl}(xs : \text{list } a), \text{ra} : \{t_1 : \text{sgl}(x : a)\}\}$$

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

$$\text{coerce } ra, ra, \text{using } pf$$

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

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

Secondly, the instruction

$$\text{project } ra, rs$$

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

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

noting that the destination register of $\text{project}$ will have the same type as if after a $\text{case}$ operation. This instruction is thus an optimised version of case: $\text{project}$ 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 $\text{cons}$ procedure should act like the $\text{cons}$ 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 $\text{project}$: given an object of the correct type, it creates a singleton such that the object represents the singleton. Recall from Sect. 2.1.3 that a $\text{cons}$ 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 $a_1$ and the tail in $a_2$, our implementation needs to allocate a tuple and initialise the first element with the word 1 and the remainder with the contents of $a_1$ and $a_2$. The code is then as follows

$$\text{cons} :: \forall (a : \text{Set})(x : a)(xs : \text{list } a). \\
\{a_1 : \text{sgl}(x : a), \text{a} : \text{sgl}(xs : \text{list } a), \text{ra} : \{t_1 : \text{sgl}(\text{cons } x xs : \text{list } a)\}\}$$

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

$$\text{totalHead} :: \forall (a : \text{Set})(x : \text{list } a)(x' : \text{list } a). (pf : xs = \text{cons } x x'). \\
\{a_1 : \text{sgl}(xs : \text{list } a), \text{ra} : \{t_1 : \text{sgl}(x : a)\}\}$$

$$\text{coerce } t_1, a_1, \text{using } pf$$

$$\text{project } t_1, t_1$$

$$\text{load } t_1, t_1(1)$$

$$\text{jump } ra$$

The alloc instruction creates a new, uninitialised tuple with the given type in $t_1$. We then use the $\text{ldi}$ instruction to move the constant 1 into $t_2$ 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.
\(A, B, v, T := \) Set \(\mid\) Prop \(\mid\) Type \\
\(\mid x \lambda x : A. B \mid A B \mid \forall x : A. B\) \\
\(\text{Ind}(X : A)\{B\}\) \\
\(\text{Ctor}(n, A)\) \\
\(\text{Elim}(C, A)\{B\}\) \\
\(T := x_1 : A_1, \ldots, x_n : A_n\)

**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 \(τ_1\) will have the singleton type \(\text{sgl}(x : a)\); 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 types 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 (\(\text{Ind}(X : A)\{B\}\)), along with their inhabitants (\(\text{Ctor}(n, A)\)). Elimination (\(\text{Elim}(C, A)\{B\}\)) 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 [13].

The typing judgement \(T \models A : B\) states that the term \(A\) has type \(B\) under the context \(T\), while the equivalence relation \(A \equiv_{βι} B\) states that the two terms \(A\) and \(B\) are equivalent under \(βι\)- and \(ι\)-reduction (elimination of inductive terms). Both typing and term equivalence are decidable.

### 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 \(\text{word}(3)\) will contain the integer \(3\).

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 \(\text{sgl}(v : T)\) 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 \(\text{sgl}(v : T)\) in \(uv\) where \(v\) and \(T\) are the value and type associated with the singleton. The word value \(uv\) is a reference to the *representation object* for the singleton, that is, \(uv\) 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

\[ \Sigma := (A; \tau) \quad \bar{\Sigma} \quad \Delta \quad \bar{\Delta} \quad \mathbf{R} \quad \bar{\mathbf{R}} \]

Instructions and instruction sequences

\[ I := \text{add } r_1, r_2 \quad \text{ldi } r_n \quad \text{move } r_d, r_s \quad \text{beq } r_1, r_2, \bar{\mathbf{R}} \text{ as } x \quad \text{apply } r_d, r_x, \bar{\Sigma} \quad \text{lda } r, \bar{\mathbf{R}} \quad \text{load } r_d, r_s(n) \text{ | store } r_d, r_s(n) \quad \text{alloc } r_d, \bar{\Sigma} \quad \text{pack } r_d, r_x \text{ as } x \quad \text{unpack } r_d, r_x \text{ as } x \quad \text{inject } r_d, r_x \text{ as } x \quad \text{project } r_d, r_x \quad \text{coerce } r_d, r_x \text{ using } p \]

\[ \mathbf{IS} := I; \mathbf{IS} \quad \text{case } r_d, r_x, \bar{\mathbf{R}} \quad \text{br } \bar{\mathbf{L}} \text{ | jump } r \text{ | halt } [\tau] \]

Word values

\[ uv := ?\tau \mid \mathbf{N} \mid \bar{\mathbf{L}} \mid \text{pack } v \text{ in } uv \mid \text{sgl } (v : T) \text{ in } uv \]

Heap values

\[ hv := (uv_1, \ldots, uv_n) \mid AT, [\Delta], IS \]

Heaps, register files, and programs

\[ H := \{ 0 \mapsto hv_1, \ldots, n \mapsto hv_n \} \quad \mathbf{R} := \{ r_0 \mapsto uv_1, \ldots, r_N \mapsto uv_N \} \]

\[ P := (H, R, \mathbf{IS}) \]

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 \( \tau \). 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

\[ \text{WORD} \triangleq \exists(n : \text{nat}).\text{word}(n). \]

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 \( \text{Set} \)): other logical terms can be hidden, including proof terms.

Example 4.2 The type

\[ \exists(b : \text{nat})(p : \text{if } b > 0 \text{ then } P \text{ else } Q), \text{word}(b) \]

that is, a word type with a hidden value associated 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 \( \forall T, [\Delta], \Gamma \) classifies abstracting over the logical variables in \( T \) and the type variables in \( \Delta \). The term \( \Gamma \) 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 \( t_1 \) and preserves the word in \( t_2 \), returning to the label in \( ra \) is

\[ \forall(x : \text{nat}), [\alpha : \mathbf{B}], \{ t_1 : \text{word}(x), t_2 : \alpha \}, \]

\[ ra : \{ t_1 : \text{word}(x + x), t_2 : \alpha \} \]

where the logical argument is \( x : \text{nat} \), the type argument is \( \alpha \), and the register arguments are \( t_1 \) with word type \( \text{word}(x) \), \( t_2 \) with parametric type \( \alpha \), and the return register \( ra \) with the code type \( \{ t_1 : \text{word}(x + x), t_2 : \alpha \} \).

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

\[ P \iff P' \]

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

\[ \Sigma = (A; \bar{\tau}) \]

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 \( \Sigma \) and \( \Sigma', \Sigma \models \Sigma' \) 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.
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

\[ v[\tilde{x} := \tilde{A}]_0[\tilde{\alpha} := \tilde{\tau}] \]

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

\[ v[T := \tilde{A}]_0[\Delta := \tilde{\tau}] \supseteq v[\tilde{x} := \tilde{A}]_0[\tilde{\alpha} := \tilde{\tau}] \]

where \(T = (\tilde{x} : \tilde{B})\) 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:

\[
\text{hn1} \circ [\text{nat}, [1, 2, 3]]
\]

and

\[
\text{hcons} \circ [\text{nat}, [1, 2, 3]]
\]

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

\[
\text{refl_equal} (\text{list nat}) [1, 2, 3] : [1, 2, 3] = [1, 2, 3]
\]

Because _case_ will branch to _hcons_, the substitution will be performed, resulting in the instruction sequence

\[
\begin{align*}
& \text{load } x_1, x_1(1) \\
& \text{apply ra, ra, (1, [2, 3], refl_equal (list nat) [1, 2, 3])} \\
& \text{jump ra}
\end{align*}
\]

where \(x, xs, \) and _pf_ have been substituted accordingly.

**Definition 5.2 (Application)** The judgement

\[
\vdash v @ \Sigma \triangleright IS
\]

holds when substituting the arguments in \(\Sigma\) into the code value \(v\) results in the instruction sequence \(IS\).

\[
\vdash (\Lambda \gamma, [\Delta], IS) \circ (@(\tilde{A} : \tilde{\tau}) \triangleright (IS[\gamma := \tilde{A}][\Delta := \tilde{\tau}]))
\]

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

\[
\begin{align*}
& \vdash H l = v \quad \vdash v@\Sigma \circ \Sigma' \triangleright IS \\
& H \vdash \mathcal{L} \circ \Sigma' \triangleright IS
\end{align*}
\]

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 (Def. 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 _ww_, where _ww_ 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.

\[
\text{case } t_1, a_1, \{ \text{hNil } @ (a, xs), \cr \text{hCons } @ (a, xs) \} \cr
\]

If the list associated with the type of \(a_1\) is \(\text{nil}\), then case will branch to \(\text{hNil}\); otherwise case will branch to \(\text{hCons}\). In the former case, the only logical argument generated is the proof \(pf : \text{xs } = \text{nil}\) while \(\text{hCons}\) gets the arguments from the \(\text{cons}\) constructor, namely \(x\) and \(\text{xs'}\), along with the proof \(pf : \text{xs } = \text{cons } x \text{ xs'}\). 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

\[
\text{sgl } (v : T) \text{ in } wv
\]

where

\[
v \equiv_{\beta_i} \text{Ctor}(n, A) B_1 \ldots B_m
\]

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

\[
v = \text{Ctor}(n, A) B_1 \ldots B_m
\]

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

\[
wv = l @ \Sigma
\]

for some label \(l\) and argument sequence \(\Sigma\). 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

\[
\text{refl_equal } T \quad v : \quad v = v
\]

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 \(\text{pack } v\) 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 \(t_1\) and \(t_2\) with the result in \(t_1\)) by

\[
\text{unpack } t_1, t_1 \text{ as } n_1 \cr \text{unpack } t_2, t_2 \text{ as } n_2 \cr \text{add } t_1, t_1, t_2 \cr \text{pack } t_1, t_1 \text{ as } \text{WORD hiding } n_1 + n_2
\]

**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 \(\text{Set}\). 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

\[
T; \Delta \vdash \tau :: \kappa
\]

holds when type \(\tau\) is well-formed with kind \(\kappa\) under the logical context \(T\) and type environment \(\Delta\).

**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

\[
T; \Delta \vdash \tau @ \Sigma \triangleright \sigma
\]

holds whenever the application of the arguments in \(\Sigma\) to the type \(\tau\) is valid and results in the type \(\sigma\), under the logical context \(T\) and type context \(\Delta\). We extend this to extended labels and heaps.
Figure 6. Singleton static semantics judgements (not including instruction sequences). We construct CiC natural numbers from meta-logical numbers using \([n]\).
Consider again Example 5.1, where we apply Example 6.1 only if the logical arguments are of the expected logical type, and the empty argument sequence to be applied to any type. Otherwise, logical arguments `nat` to apply `nat` a noting that the first argument, `nat`We shall call this type `xs` of natural numbers `[1` represented using a word type boxed by a tuple. By singleton types for those constructor arguments with informative logical context

**Definition 6.3 (Representation types)** The judgement

\[
\Psi \vdash \tau \quad \Gamma; \Delta \vdash \tau \ominus (\Sigma_1 \ominus \Sigma_2) \triangleright \sigma
\]

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`

\[
\forall (a : \text{Set})(xs : \text{list } a).
\{a_1 : \text{sgl}(xs : \text{list } a),
ra : \forall (x : \text{list } a)(xs' : \text{list } a)
(pf : xs = \text{cons } x xs').\{t_1 : \text{sgl}(x : a)\}\}
\]

We shall call this type `\(\tau_{\text{head}}\)` in the following.

To apply `nat`, we must show

\[\Gamma \vdash \text{nat} : \text{Set}\]

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

\[
\forall (zs : \text{list } \text{nat}).
\{a_1 : \text{sgl}(zs : \text{list } \text{nat}),
ra : \forall (x : \text{list } \text{nat})(xs' : \text{list } \text{nat})
(pf : xs = \text{cons } x xs').\{t_1 : \text{sgl}(x : \text{nat})\}\}
\]

To apply `[1, 2, 3]`, we must show

\[\Gamma \vdash [1, 2, 3] : \text{list } \text{nat}\]

as `zs` has type `\text{list } \text{nat}`. Again this holds, and so, letting `\(\tau_{\text{head}}\)` be

\[
\{a_1 : \text{sgl}([1, 2, 3] : \text{list } \text{nat}),
ra : \forall (x : \text{list } \text{nat})(xs' : \text{list } \text{nat})
(pf : [1, 2, 3] = \text{cons } x xs').\{t_1 : \text{sgl}(x : \text{nat})\}\}
\]

we can derive

\[\Gamma; \Delta \vdash \tau_{\text{head}} \circ \circ \circ (\text{nat}, [1, 2, 3]) \triangleright \sigma_{\text{head}}.\]

### 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

\[\Gamma \vdash v \downarrow \tau\]

holds when the value `v` is represented by the type `\(\tau\)` under the logical context `\(\Gamma\)`.

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 `\text{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

\[\Gamma \vdash [1, 2, 3] \downarrow \langle \text{word}(1), \text{sgl}(1 : \text{nat}), \text{sgl}(2, 3) : \text{list } \text{nat} \rangle\]

which matches our description from Sect. 2.1.2.

### 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` constructors, 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

**Inductive** `\(T : \text{Set} := \ldots\)`

the elimination candidate for `\(v, T, n\)` is

\[
elims v T n = (\text{Ctor}(n, T) x_1 \ldots x_m, \langle x_1 : A_1 \rangle \ldots \langle x_m : A_m \rangle)
\]

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 `\(\text{list } T\)` and some value `\(v\)` are

- `\(\text{elims } v \langle \text{list } T \rangle 0 = (\text{nil}, e)\)`
- `\(\text{elims } v \langle \text{list } T \rangle 1 = (\text{cons } a as, (a : T)(as : \text{list } T))\)`
- `\((p : v = \text{cons } a as)\)`

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 `\(\text{listP } T P\)` and some value `\(v\)` are

- `\(\text{elims } v \langle \text{listP } T P \rangle 0 = (\text{nilP}, e)\)`
- `\(\text{elims } v \langle \text{listP } T P \rangle 1 = (\text{consP } a p as, (a : T)(p : P a))\)`
- `\((as : \text{listP } T P)\)`
- `\((q : v = \text{consP } a p as)\)`

where `a`, `as`, `p`, and `q` are fresh. Note that, unlike the representation type for `\(\text{listP}\)` (see Eg. 6.3), the elimination candidates are effected by the extra proof term: the environment portion of `\(\text{elims } v \langle \text{listP } T P \rangle 1\)` includes the proof `\(p\).`
The typing rules relating to singleton operations rely on the auxiliary judgements from Sect. 6.3. The \texttt{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 \texttt{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 \texttt{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 \texttt{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 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 \texttt{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 \texttt{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. \cite{Hamid:2007}, 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 \( \Psi \) and register file type \( \Gamma \), 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 \cite{Singleton:Coq} 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 \( \Delta \vdash \sigma \) then

\[
\vdash hv \sigma \Gamma \vdash hv' \text{ for some } hv'.
\]

We note that all CiC terms in \( hv' \) are well-formed under the context \( \Gamma \).

### Lemma 2

If \( \Psi \vdash hv : \tau \), and \( \Delta \vdash \sigma \Gamma \vdash \Gamma' \), and

\[
\vdash hv \sigma \Gamma \vdash IS \text{ then } \Psi \vdash IS \vdash \Gamma'.
\]

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

### Lemma 3 (Progress)

If \( \vdash (H, R, I) \) then \( I = \text{halt} \tau \) or \( (H, R, I) \rightarrow P \) for some \( P \).

\footnote{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 \textit{list} \( T \).}
\[ \Gamma r_1 = \text{word}(n_1) \quad \Gamma r_2 = \text{word}(n_2) \]
\[ \begin{array}{c}
\Gamma \Delta \Psi; \Gamma(r_d \mapsto \text{word}(n_1 + n_2)) \vdash IS \\
\end{array} \]
\[ \begin{array}{c}
\Gamma; \Delta; \Psi; \Gamma \vdash \text{add } r_d, r_1, r_2; IS \\
\end{array} \]
\[ \begin{array}{c}
\Gamma; \Delta; \Psi; \Gamma \vdash \text{move } r_d, r_s; IS \\
\end{array} \]
\[ \begin{array}{c}
\Gamma; \Delta; \Psi; \Gamma \vdash \text{store } r_s, r_d(n); IS \\
\end{array} \]
\[ \begin{array}{c}
\Gamma; \Delta; \Psi; \Gamma \vdash \text{load } r_d, r_s(n); IS \\
\end{array} \]
\[ \begin{array}{c}
\Gamma; \Delta; \Psi; \Gamma \vdash \text{load } r_d, r_s(n); IS \\
\end{array} \]
\[ \tau = \exists (x : T). \xi \quad \text{and } \forall \theta \subseteq \Gamma \]
\[ \begin{array}{c}
\Gamma; \Delta; \Psi; \Gamma \vdash \text{jump } r \\
\end{array} \]
\[ \begin{array}{c}
\Gamma; \Delta; \Psi; \Gamma \vdash \text{beq } r_1, r_2, L \quad \text{as } p; IS \\
\end{array} \]
\[ \begin{array}{c}
\Gamma; \Delta; \Psi; \Gamma \vdash \text{lda } r, L; IS \\
\end{array} \]
\[ \begin{array}{c}
\Gamma; \Delta; \Psi; \Gamma \vdash \text{ida } r, n; IS \\
\end{array} \]
\[ \begin{array}{c}
\Gamma; \Delta; \Psi; \Gamma \vdash \text{apply } r_d, r_s, \Sigma; IS \\
\end{array} \]

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

**Lemma 4** (Preservation) \( \vdash P \) and \( P \iff P' \) 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) \( \vdash P \) and \( P \iff P' \) 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 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 \( \lambda_{\text{tl}} \), 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 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 forTL 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
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 $f\ f\ v$ instead of $\text{sgl}\ t\ f\ v$, 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 \cite{4} takes a similar approach to TL, using linear LF \cite{3} as the assertion logic rather than CiC; see \cite[Sec. 7]{4} 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 LTT 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. \cite{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 \cite{11,13,5,12}: the type \( \{ x : T | P \} \) 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

\[
\text{Inductive refinement}(t : \text{Set})(P : t \to \text{Prop}) : \text{Set} := \}\text{refinement} I : \forall(v : t), P\ v \to \text{refinement}\ t\ P
\]

Although refinement types subsume our singleton types, in that the refinement types \( \{ x : T | x = v \} \) corresponds to the type $\text{sgl}(v : T)$, 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.

\cite{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 indices of word and array types. As with our system, these indices limit the modularity 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-forms 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.

\cite{7} give a dependently typed assembly language used as a target for their CCured tool \cite{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 Economy and the Australian Research Council through the ICT Centre of Excellence program.

### References


