Software Systems Research Group
http://www.ssrg.nicta.com.au/

NICTA

CDSL Version 1

Simplifying Verification with Linear Types

NICTA Technical Report 1833-9646-8393

Liam O'Connor, Gabriele Keller, Sidney Amani, Toby Murray
Gerwin Klein, Zilin Chen and Christine Rizkallah
firstname.lastname@nicta.com.au

2014

Abstract

We introduce a purely functional domain specific language, CDSL, which aims to substantially reduce the cost of producing efficient, verified file system code. Given an executable specification of a file system, the CDSL compiler generates C code and, when fully implemented, will also generate an Isabelle/HOL proof linking the specification and the C implementation. We present two operational semantics for CDSL: (1) a value semantics, well suited for verification, and (2) an update semantics, which can be mapped to efficient C code. We outline the equivalence proof between these two semantics and discuss how the type system guarantees properties like termination, correct error handling, absence of memory leaks and aliasing.

© 2014 NICTA

1 Introduction

Our research group aims to provide a complete set of reliable, even provably correct systems software. With seL4 [Klein et al., 2014], a comprehensively formally verified microkernel, as well as with automatic device driver synthesis from formal specifications [Ryzhyk et al., 2009], we have made significant inroads towards this aim. File systems pose an especially hard challenge, however, due to their diversity. For example, the Linux kernel source tree presently contains 49 different file systems. Moreover, the advent of new classes of storage devices, and the desire for new (and more specialised) functionality, such as different reliability-performance trade-offs, drives growth in the number of file systems. New usage scenarios, such as those resulting from the widespread adoption of virtualisation, creates pressure to change internal APIs in the storage stack [Jannen et al., 2013], which in turn requires massive re-writing of file system code. To tackle these problems effectively, we need to find a way to dramatically reduce the cost of producing correct code.

At the core of our approach are several observations about real-world file systems: While the use of static analysis for finding bugs in low-level systems is useful in many cases [Ball et al., 2010, Bessey et al., 2010], it is too limited for file systems code, as the majority of file system bugs are deeper semantic faults [Lu et al., 2013], which are not usually amenable to static analysis. Therefore, to ensure reliable file systems, we use formal proof to demonstrate that the compiled binary behaves according to a compact high level functional specification, which captures the intended semantics of file system operations.

Furthermore, we observe that real-world file systems code has a highly modular structure. This, if exploited carefully, can make verification easier and also facilitate re-use. Moreover, the storage stack of contemporary operating systems requires and defines many intermediate abstraction levels, and while, for each module, the translation between abstraction levels is logically straightforward, it is obscured by error handling with tricky corner cases. As a result, correctness proofs at this level are, to a large degree, mainly tedious, but not conceptually challenging.

Code and proofs with these properties are ideal candidates for generation from higher-level domain specific languages. For our framework, we implemented two distinct domain specific languages for two separate tasks: a data description language, DDSL, from which we generate code for serialisation and de-serialisation of data and accompanying correctness proofs, together with a second language, CDSL, which we use to write the executable specification of the file system control code, again generating code and correctness proofs. In this paper, we discuss the latter language.

The design of CDSL is strongly guided by the experience we gained with the seL4 microkernel verification project. This project showed what the executable specification and generated C code should look like, how to model different aspects of the execution, and which program properties can and should be ensured statically, so that they can be used to generate the corresponding Isabelle/HOL proofs.

A central feature of CDSL is its linear type system. It not only helps us enforce these properties statically, but it also affects the formalisation of the operational semantics. We define two semantic interpretations for CDSL programs and prove them equivalent in Isabelle/HOL. The first is a purely-functional value semantics, which greatly assists in manual proofs of refinement from the functional high-level specification to the executable specification, because it abstracts over low level details like destructive update and heap allocation. The second is the update semantics, which is less abstract, and serves as a basis for the refinement proof to the generated C code. We are currently working on generating this C refinement proof fully automatically.

To summarise, the contributions of this paper are the following:

1. We present the domain specific language CDSL for the low level specification of file systems and similar code.
2. We formalise the static semantics, as well as the big-step dynamic semantics for both semantic interpretations of CDSL.
3. We discuss how the strong static semantics of this language imply a set of desirable properties for the generated code, and how we prove those properties.
4. We prove in Isabelle/HOL that the value semantics and update semantics of the language are equivalent and present an outline of the proof. To the best of our knowledge, this is the first formal proof of such an equivalence for a language with a linear type system.

The remainder of the paper is structured as follows: In Section 2, we give an overview of our file system framework, in particular CDSL's role, and give details on the current state of the project. Section 3 introduces CDSL informally and discusses the implications of using linear types for verification and implementation. Section 4 gives a formal definition of CDSL's core language, its static semantics as well as both dynamic semantics, and outlines the key safety and equivalence proofs. Section 5 briefly discusses how refinement proofs can be generated from CDSL code, and we conclude with a discussion of related and future work.

2 Overview

To implement a fully trustworthy file system, we start verification from a high-level correctness specification, which abstracts over all aspects of the file system execution, for instance describing it simply as a map from inode numbers to their contents. We connect this specification via formal proofs all the way down to the executable. In our framework, we split this process into several steps: (1)

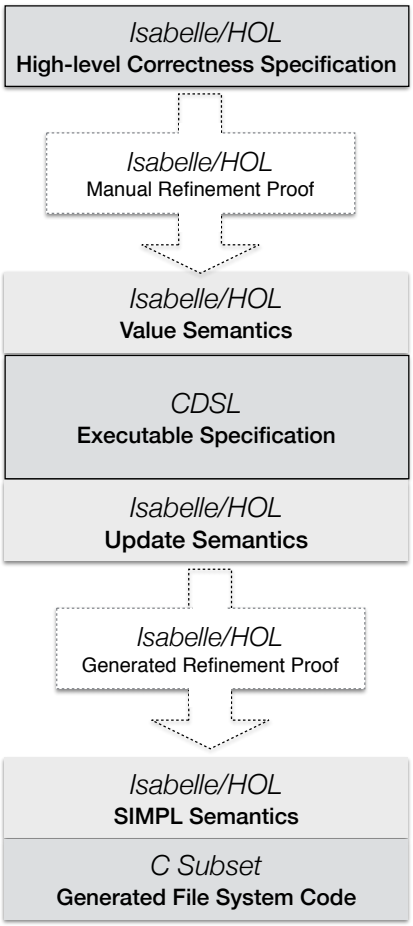

Figure 1: Structure of a typical correctness proof with CDSL

we connect the high-level specification via refinement proofs to an executable specification, (2) we show that the C implementation is correct with respect to the executable specification, and (3) we prove that the machine code produced by the C compiler is correct.

To reduce the cost of the whole verification process, we looked at each step separately: Getting from the high-level specification to an executable specification that can be used as a basis for an efficient implementation involves significant expert knowledge, making it unlikely that we can gain much by trying to automate Step (1). Therefore, we reduce the costs of this step by careful modular design, which maximises re-use of components across different, but related file systems.

Connecting the executable specification and the C code for Step (2) is a different story. This specification is algorithmically close to regular file system code, but has a much richer static semantics. Even though it still abstracts over some issues, like memory management, mapping it to C code is relatively straightforward. This makes Step (2) an ideal candidate for automatic generation, both of the C code and the proofs connecting the C code to the specification.

Step (3), the verification of the binary generated by the C compiler with respect to its formal semantics will be handled using a technique developed for the verification of the seL4 binary [Sewell et al., 2013]. To apply this technique, we have to restrict ourselves to using only a particular subset of C when generating code from CDSL. It is no coincidence, however, that this subset is a natural fit for CDSL, as it excludes exactly those features of C which are semantically problematic anyway.

In Keller et al. [2013], we gave an high-level overview of our framework and discussed Step (1) in more detail. In the remainder of this paper, we only look at Step (2). More precisely, we introduce CDSL, the language we designed and implemented to express the executable specification.

As described above, CDSL plays multiple roles: (a) it is an interface to the manual correctness proofs of Step (1). As such, the more abstract its semantics, the better; (b) it is the source for generating efficient C code. A task which is made easier the more low-level the semantics of CDSL are. Finally, (c) it is also the source for generating correctness proofs, which calls for an expressive static semantics.

We resolve the contradicting requirements of Point (a) and Point (b) by defining two dynamic semantics for CDSL: a high-level value semantics and a more low-level update semantics. We formally proved that these two are equivalent, allowing us to pick the semantics best suited for the task at hand. CDSL's strong static semantics, a linear type system, is the basis for connecting these two dynamic semantics and facilitates Point (c).

Figure 1 illustrates how Step (1) and Step (2) connect in our framework and the role CDSL's update and value semantics play in this process: the file system developer provides a high-level functional specification in Isabelle/HOL and a executable specification

written in CDSL. The CDSL program's value semantics can be automatically formalised in Isabelle/HOL, according to the rules we present in Section 4. The developer has to prove manually that this formalisation is a refinement of the high-level specification. The CDSL compiler generates C code based on the update semantics of the CDSL program, and a proof showing that the generated C code is equivalent to that semantic interpretation. The equivalence proof of the value and update semantics presented in this paper completes the proof chain from high-level specification to C code.

At the start of this project, we analysed existing file system code to get a better handle on Step (2). We found that the code generally fits into one of three major categories: (i) data serialisation and de-serialisation code; (ii) code creating and maintaining data structures, such as lists, red-black trees, and so on; and (iii) the remaining control code.

Data serialisation code (i) is very common, not just in systems programming, and it is well known that the code is particularly tedious and error prone. Not surprisingly, there are many different tools and DSLs to facilitate this task and generate code from a more abstract data description, for example McCann and Chandra [2000], Back [2002], Fisher and Walker [2011], to name just a few. Building on these ideas, we implemented a data description language, DDSL, which not only generates the code (CDSL in our case) from the specification of the on-disk and in-memory representation, but also the high-level specification and the correctness proofs for data serialisation and de-serialisation routines. We discuss the features of DDSL in more detail in Keller et al. [2013].

Data structure code (ii) can mainly be packaged up as abstract data types (ADTs) and for now, we simply assume the existence of verified implementations of the ADTs we need. Examples of such functional data container verifications already exist [Lammich and Lochbihler, 2010, Lochbihler, 2013].

In a realistic file system, it is not always possible, primarily for performance reasons, to maintain abstraction over the representation of data when dealing with data structures. For the design of this domain specific language, however, we do not aim at 100% coverage at all costs. Instead, we try and keep the language as simple as possible and accept that at a select few places in the code, it may become necessary to replace an inefficient, automatically generated and verified portion of the code by a more efficient hand written, manually verified version.

Finally, we use CDSL to express an executable specification of file system control code (iii) from which the compiler generates C code and for which we aim to also produce a correctness proof showing that the C code implements the CDSL semantics. The control code in file systems has a fairly simple structure and large amounts of it are concerned with error checking and appropriate recovery. Error handling is a significant source of bugs[1]. This suggests that we should statically ensure errors are properly handled, as neither testing nor the various static checkers available for C are a reliable way to guarantee this otherwise. Static checking is tractable due to the absence of recursion or general loops outside of data structure traversals. In fact, in all of the seL4 microkernel, there is only a single such loop outside this pattern.

By requiring the ADTs to come with appropriate traversal operations, we can avoid having general loops or recursion in CDSL at all, resulting in a strongly normalising language where termination is a given. Again, this restriction in the language may mean that for some file systems, some minor manual verification and coding effort is necessary, as for one or two loops, it might not be possible to express them in CDSL.

We designed and developed the two domain specific languages, DDSL and CDSL, hand in hand with implementation of a first case study in this framework: a simple, but realistic flash file system [Keller et al., 2013]. This hands-on approach resulted in a rapid feedback loop, which was essential for us to obtain an understanding of what features are possible and necessary; in particular, since our group combines —and the problem calls for— people from different fields of expertise: systems, formal methods and programming languages. We are continuing to evaluate and refine the framework in a second, more complex file system case study.

3 CDSL

CDSL is a purely functional language because referential transparency drastically simplifies verification. Unfortunately, most purely functional languages are unsuitable for the implementation of file systems, as they rely on automatic garbage collection for memory management, and favour copying of data over inplace updates, which introduces performance overheads. By using linear types for all dynamic data structures, we support a functional semantics and memory safety in the presence of destructive updates, without garbage collection [Wadler, 1990]. Programs in CDSL can be understood through two semantic interpretations: The value semantics is a functional view, passing arguments by value; whereas the update semantics is a more imperative view, with a mutable store, pointers, and destructive updates. A formal description of both semantics and a proof of their equivalence is given in Section 4.

Linear types have proven to be difficult to integrate into general purpose languages. In our particular case, however, the trade-off is quite different. Because of the domain specific nature of CDSL, the restrictions that linear types impose are acceptable. For example, we do not need to be able to express data structures containing sharing, as we abstract over their concrete representation in CDSL and implement them outside the language.

In CDSL, we intentionally leave out most of the features which make functional languages so powerful: it does not support partial application, polymorphism, user-defined higher-order functions, or even recursion! For a low-level specification, none of these features are necessary. The absence of the first three simplifies correctness proofs and facilitates efficient code generation. Without recursion, termination becomes straightforward to prove.

The basic idea behind linear types is not complicated: any object of linear type has to be used exactly once. As a consequence, any function written in CDSL must return the same number of values of a specific linear type as it consumes as arguments, unless it

[1] Error handling uses idiomatic code, not unlike that which lead to the infamous Apple SSL @goto fail@ bug (http://avandeursen.com/2014/02/22/gotofail-security/).

internally calls a built-in function which creates a new value of linear type or destroys one. With respect to the implementation, this means that we can re-use the heap space of an object as soon as it is used, because the type checker already checked that it is not accessed a second time. Furthermore, unless a built-in destructor is called, we need (and can) not free the space, as the object will definitely be used again. This way, we get the efficiency of destructive updates with the guaranteed absence of any memory leaks.

It would have been possible to model the update semantics with a state monad, which, looked at in isolation, is arguably more convenient to use. The problem with this approach is that file system code typically maintains a huge state, but subcomponents of the code often only affect a tiny portion of that state. This can be easily modelled using linear types: the required parts of the data structure can be taken in isolation, altered, and then re-inserted, and this process can be arbitrarily fine grained. Encoding the same level of detail in the type system with state monads would require combining many different sub-state monads, which results in awkward, hard to maintain code, particularly in a first order language, and complicates verification.

In the following, we will look at some sample code and discuss the effect of linear types in CDSL, both from the programmer's point of view as well as with regard to the consequences for its compilation. The basic handling of linear types, including **let!**, is similar to Wadler [1990], but the particular way we handle errors and deal with records is novel.

First Examples. In CDSL, values may either have unboxed type or boxed type. Values of unboxed types (written as $t^\sharp$) are always passed by copying. Values of boxed type are represented as pointers in the update semantics, and may be of linear type (t^1) or shareable type (t^∞).

The following declaration, for example, states that *objNew* takes a regular integer as input, and returns a linear value of type `Object`, where `Object` is defined elsewhere.

$$\begin{aligned} &\mathit{objNew} :: \texttt{Int}^\sharp \to \texttt{Object}^1 \\ &\mathit{objFree} :: \texttt{Object}^1 \to () \end{aligned}$$

Similarly, *objFree* takes a linear object, but returns no results. Now, in the value semantics, the type of *objFree* means that it must simply return nothing and perform no side-effects, and would, consequently, be pretty pointless. In the update semantics, however, *objFree* consumes the input object, without returning a new object in its place, so in effect, it frees the memory associated with the object. Therefore, while it does not return a meaningful value, it can be seen as having a side-effect. Despite this, linear types make sure that any type correct program is still referentially transparent.

Functions in CDSL may take any number of parameters and return any number of results. In the following CDSL code snippet, both *objNew* and *objFree* are built-in primitives.

$$\begin{aligned} &\mathit{objNew} :: \texttt{Int}^\sharp \to \texttt{Object}^1 \\ &\mathit{objFree} :: \texttt{Object}^1 \to () \\ &\mathit{foo} :: (\texttt{Int}^\sharp, \texttt{Int}^\sharp) \to (\texttt{Object}^1, \texttt{Object}^1) \\ &\mathit{foo}\,(a,b) = \textbf{let}\ \mathit{obj}_1 = \mathit{objNew}\ a \qquad \text{— (1)} \\ &\qquad\qquad\ \ \mathit{obj}_2 = \mathit{objNew}\ b \qquad \text{— (2)} \\ &\qquad\qquad\ \ \mathit{obj}_3 = \mathit{objNew}\,(a + b) \quad \text{— (3)} \\ &\qquad\qquad\ \ \_ = \mathit{objFree}\ \mathit{obj}_2 \qquad \text{— (4)} \\ &\qquad\ \ \textbf{in}\ \mathit{obj}_1,\ \mathit{obj}_3 \end{aligned}$$

This program is valid, since obj_1, obj_2 and obj_3 are all used exactly once. Leaving out (4) would lead to a type check error, as obj_2 is never used, as would inserting a clause $\_ = \mathit{freeObj}\ \mathit{obj}_1$, since obj_1 would now be used twice.

We could relax our constraints and just require each linear object to be used at most once (giving an affine type system), simply assuming that any unused object is not required any more and automatically freeing it, which would be more convenient. On the other hand, if the programmer accidentally omits a newly created object from the return of the function, automatically freeing it without any compiler warning could lead unexpected behaviour, so for now we have decided on explicit calls to the destructor.

Let! Linear types are great to indicate where we can perform in-place updates, but if we only want to look at an object of linear type and not replace it with an altered version, this can lead to unnecessarily complicated code. Consider a function *sizeofObj'*, which returns some information about a linear object. It would have to return a "new" object and be of type:

$$\mathit{sizeofObj'} :: \texttt{Object}^1 \to (\texttt{Int}^\sharp, \texttt{Object}^1)$$

because passing the object to the function means that we cannot access it anywhere else, even though we know that the object in the argument is exactly the same as the object in the result. To eliminate this awkwardness, we borrow a feature from Wadler [1990], where a new binding form $\textbf{let!}\ (v)\ w = \mathit{expr}_1\ \textbf{in}\ \mathit{expr}_2$ is introduced. The value of $expr_1$ is bound to w in $expr_2$, as in a regular *let*-expression, except that we can temporarily use the value of linear type v in $expr_1$ freely as a shareable type for reading, and then once again view it as a linear type in $expr_2$. Therefore, we can have

$$\mathit{sizeofObj} :: \texttt{Object}^\infty \to \texttt{Int}^\sharp$$

and call it as follows:

```
foo :: Object¹ → ...
foo obj = let! (obj) size = sizeofObj obj
          in ...
```

The types of variables bound in a **let!** are restricted to avoid the unintentional introduction of aliasing (see Section 4 and Wadler [1990] for a more detailed discussion of this problem).

Records. Field access for records and product types in a linear type system can be subtle. We want to make sure it is semantically clean and supports an efficient implementation.

Assume the type `Object` is a type synonym for a record type, which in CDSL are anonymous and structural:

```
Object = { size :: Int♯, f1 :: T¹, f2 :: T¹}
```

Let us say we want to extract the field f_1 from an object of type `Object`^{1}. What is the result of the extraction? If we return just a value of type `T`^{1}, it would mean we lose the object itself, because it has already been used, so we clearly have to return a new object. At the same time, we can not simply return a reference to the old object and one to the field f_1 without introducing aliasing. We cannot copy the field value, as the type `T`^{1} may be arbitrarily large. Our solution is to return an object where the field f_1 may not be extracted again, and reflect that in the type, which in this paper we typeset as strikethrough. So, applying the extraction, **take** in CDSL, to *obj* :: `Object`^{1} with field selector f_1 yields two values with type:

```
take obj.f1 : ({ size :: Int♯, f1 :: ~~T¹~~, f2 :: T¹}, T¹)
```

Conversely, we can re-insert a field into a record where this field has been taken using **put**, as in the following code snippet, which swaps fields f_1 and f_2 in an object:

```
obj1, v1 = take obj.f1
obj2, v2 = take obj1.f2
obj3     = put obj2.f2 := v1
obj4     = put obj3.f1 := v2
```

We could view an object with a taken field as supertype of an object with no taken fields. For example, a function that only ever looks at f_1 could accept an object with a taken f_2 field. The current version of CDSL does not support it, and it has not been necessary in our case study so far, but there are arguably use cases where subtyping would make CDSL more convenient to use, so this is an issue worthwhile looking into in future.

Variants. In addition to records (product types), CDSL also has support for variant types or tagged unions (sum types). For example, the type $\langle A_1 :: T, A_2 :: U\rangle^1$ is a linear value that is either of type `T` or of type `U`, depending on the tag (either A_1 or A_2). Pattern matching on such values is performed via **case** expressions:

```
Object = ⟨ A1 :: T, A2 :: U ⟩

convert :: U¹ → T¹

example :: Object¹ → T¹
example o = case o of
              A1 t → t
              A2 u → convert u
```

Error Handling. Large swathes of the code written in CDSL is designed to accommodate recovery and reporting from errors. In addition, a large source of bugs in file systems and other systems code is improper handling of error conditions. We therefore designed CDSL to include a robust error handling system, in which expressions that may result in errors are given a special type:

```
f :: T¹ or Fails U¹
```

Here, *f* will return a value of type `T`^{1} if it succeeds, or `U`^{1} along with an error code if it fails. The type *t* `or Fails` *u* could therefore be considered similar to this Haskell data declaration:

```
data OrFail t u = Success t
                | Failure ErrorCode u
```

A function may fail by returning **fail** *errCode* $v_1, \ldots$ rather than simply returning a collection of values as normal. We also include a built in destructor for these error types, called **handle**, where

```
let x = expr1 handle (errCode, v1, ...) expr2
in expr3
```

would be equivalent to the Haskell

```
let r = expr1
in case r of
     Success x          → expr3
     Error errCode v1 ... → expr2
```

We can now give a more accurate type to our *objNew*, which takes into account the possibility of an out of memory error:

```
objNew :: Int♯ → Object¹ or Fails ()
```

Consider a function *mergeObj* which merges two objects of linear type into a new object, destroying the argument objects in case of success. If this function, in case of failure, leaves the initial objects intact, its type is:

```
mergeObj :: (Object¹, Object¹)
         → Object¹ or Fails (Object¹, Object¹)
```

The following example function creates two objects and returns the merged object. Each of the functions it calls may fail, so the type system ensures that, in the error case, all objects created in the meantime are correctly freed:

```
example :: Int♯ → Object¹ or Fails ()
example size = let
  obj1 = objNew size
         handle code (fail code)
  obj2 = objNew (2 * size)
         handle code let _ = objFree obj1
                     in fail code
  obj3 = mergeObjs (obj1, obj2)
         handle (code, obj1', obj2')
           let _ = objFree obj1'
               _ = objFree obj2'
           in fail code
in obj3
```

Code like this is common in file systems and, modulo the syntax, quite similar to what we would write in C. The difference is that in CDSL, the type system ensures the errors are handled, and linear types ensure that everything which has been allocated and is not needed any more is appropriately freed. We could, in fact, easily add syntactic sugar which would free the programmer from writing the error handling code if it does nothing but freeing objects and passing the error code. We intend to implement such improvements further into the development of our framework.

Loops. While CDSL does not support recursion or general loops, it does support loops with straightforward termination measures, such as traversals of data structures. We define a language of iteration schemes, loop patterns that are known to terminate, and apply them to a loop body using **for** expressions. For example, a linear array type, say $[\texttt{Int}^\sharp]^{\mathbf{1}}$, can be updated element-wise using the built-in iteration scheme `map`:

```
incAll :: [Int♯]¹ → [Int♯]¹
incAll as = for map(as) : a → a + 1
```

As the loop body may be called multiple times, the loop body cannot reference linear variables from outer scopes, because they may not be usable for each iteration of the loop. Accumulators can be added to thread state through loops using the built in `acc` operator. For example, we could compute the sum of the numbers in the array while we update them:

```
incSum :: [Int♯]¹ → (Int♯, [Int♯]¹)
incSum as = for map(as) acc 0 : a, acc →
              a + 1, a + acc
```

To simply compute the sum, though, we do not need to update the array at all. For that, we can use the iteration scheme `fold`, which operates on shareable arrays:

```
sum :: [Int♯]^∞ → Int♯
sum as = for fold(as) acc 0 : a, s → a + s
```

In these examples, we have used the built-in iteration schemes for arrays, but abstract data types may also provide their own traversal schemes.

Our implementation of CDSL also supports a number of other syntactic conveniences, which have relatively uninteresting semantics. We do not discuss them here primarily due to space constraints, but they are included in the Isabelle/HOL formalisation.

Variable names	x, y		
Function names	f, g		
Field names	a, b		
Expressions	e	::=	$\overline{x}$
		|	**fail** $x\ \overline{x}$
		|	$f(\overline{x})$
		|	**bind** $e \Rightarrow B$
		|	**bind!** $(\overline{x})\ e \Rightarrow B$
		|	**if** x **then** e **else** e
		|	**take** x.a
		|	**put** x.a $:= y$
Binders	B	::=	$\overline{x}.\ e$
		|	**handle** $x\ \overline{x}.\ e$
		|	$\overline{x}.\ e$ **handle** $y\ \overline{y}.\ e'$
Return Types	T	::=	$\overline{\tau}$ | `Fails` $\overline{\tau}$
		|	$\overline{\tau}$ `or Fails` $\overline{\rho}$
Types	τ, ρ	::=	$t^{\mathbf{1}}$ | t^{∞} | $t^{\sharp}$
Base Types	t	::=	$\{\overline{\mathrm{a} :: \tau_?}\}$
		|	`Int` | `Bool` | `Err` | $\cdots$
Record Field Types	$\tau_?$	::=	τ | $\not\tau$

Figure 2: The core fragment of A-normal CDSL

4 Formalisation of CDSL

We have formalised the full CDSL language in Isabelle/HOL, and proven all type-safety and equivalence theorems against that formalisation. As these formalisations and proofs consist of several thousand lines of Isabelle code, we restrict ourselves to the most interesting fragment of the language in this section, and provide only an outline of the proofs we have completed.

The grammar of CDSL is given in Figure 2. As opposed to the full formalisation in Isabelle/HOL, this fragment is A-normalised [Sabry and Felleisen, 1992] and we omit several features, which are not significant to the results we have proven.

In addition, **let** and **let!** expressions are replaced by **bind** and **bind!** expressions respectively, to accommodate formalisation of CDSL's error handling features. The only difference is that **bind** syntax is in the opposite order to **let**. In **bind**, the expression being bound appears first, followed by a binder, which is an abstraction that may contain an error-handling alternative. For example, the expression **let** $x = e$ **handle** $y.\ e_f$ **in** e_s is equivalent to **bind** $e \Rightarrow x.\ e_s$ **handle** $y.\ e_f$.

To accommodate the separately verified abstract data types, we have made our formalisation extensible with additional types, values, and functions. For our formalisation, we do not consider a whole CDSL program (a series of function definitions), but rather expressions denoting a single function. Function calls may refer to a function defined in CDSL or an abstract function defined separately. The semantics does not distinguish between these two cases. As CDSL does not allow recursion, this has no impact on the expressiveness of the language.

The typing rules for the CDSL core fragment are given in Figure 3. In CDSL, all expressions may either succeed and return a collection of values ($\overline{\tau_i}$), or fail and return a code along with a collection of values (`Fails` $\overline{\tau_i}$). In addition to the types for those two cases, the two promotion rules $\leq_1$ and $\leq_2$ allow for a supertype that supports both possibilities ($\overline{\tau_i}$ `or Fails` $\overline{\rho_j}$).

As our type system supports linear types, the two structural rules of contraction and weakening are not allowed in general:

$$\frac{\Gamma, x : \tau, x : \tau \vdash e : T}{\Gamma, x : \tau \vdash e : T}\textsc{Contr.} \quad \frac{\Gamma \vdash e : T}{\Gamma, x : \tau \vdash e : T}\textsc{Weak.}$$

The absence of these rules means that we must treat the context Γ as a multiset of type judgements, not a set. Moreover, it ensures that an expression can only be well typed if it uses each variable in its context exactly once. CDSL is not a completely linear language, however: It does allow contraction and weakening for values of unboxed ($t^{\sharp}$) and shareable (t^{∞}) types.

Structural rules are awkward to use directly in mechanised formalisations, as they can be applied to any type judgement. Rather than add structural rules for these cases, we borrow a trick from Ahmed et al. [2005] and contain contraction to an explicit context-splitting relation, written $\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$, where each judgement $x : \tau$ in Γ must occur in either Γ_1 or Γ_2, or both if τ is not linear.

Similarly, we contain weakening to a relation $\Gamma \overset{\text{weak}}{\rightsquigarrow} \Gamma'$ where $\Gamma' \subseteq \Gamma$ and each linear judgement $x : t^{\mathbf{1}}$ in Γ is in Γ'.

The final structural rule present in most logics and type theories is Exchange:

$$\frac{\Gamma_1\Gamma_2 \vdash e : T}{\Gamma_2\Gamma_1 \vdash e : T}\textsc{Exchange}$$

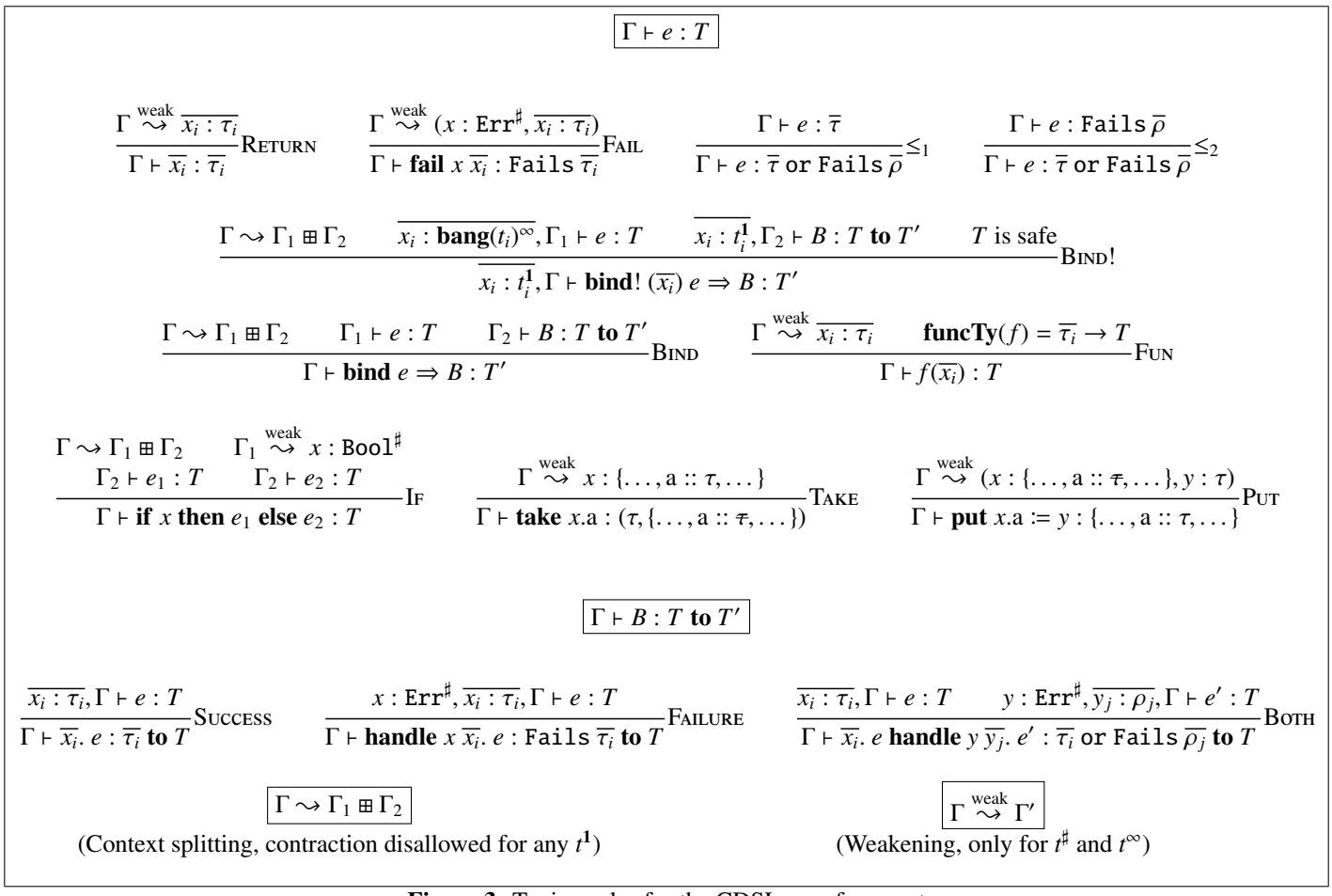

Figure 3: Typing rules for the CDSL core fragment

For the formalisation presented here, Exchange is allowed, however our Isabelle/HOL formalisation uses de Bruijn indices [de Bruijn, 1972] rather than names, which canonicalises the order of judgements. As a result, Exchange is not explicitly formalised.

The rule Bind! is of particular interest. A **bind!** expression, written **bind**! $(\overline{x_i})\ e \Rightarrow B$, allows some set of linear variables $(\overline{x_i})$ to be temporarily made shareable, as mentioned in Section 3. The type function $\mathbf{bang}(t)$ recursively replaces all linear components $t'^{\mathbf{1}}$ in t with $\mathbf{bang}(t')^\infty$ (and leaves all other types alone). It is crucial to ensure statically that none of the values $\overline{x_i}$ are part of the result of evaluating e and bound in B — Otherwise, aliasing will occur between the returned shareable value and the linear value that is available in B. To achieve this, we, like Wadler, use a type-based approximation. We say that the bound type T is safe iff it shares no shareable components t^∞ with the types of the shareable values $\overline{\mathbf{bang}(t_i)^\infty}$.

4.1 Value Semantics

Figure 4 contains the dynamic value semantics of the CDSL fragment. The big-step evaluation relation $\gamma \vdash e \Downarrow r$ states that the expression e evaluates to a return value r under the value environment γ. In many ways, the semantics is entirely typical of a purely functional language. One difference is the way function calls are handled. As mentioned, function calls may refer to functions defined in CDSL, or to abstract functions verified separately. We associate with all functions f a semantics $[\![f]\!]$ which is a relation from input values to output return values. Because recursion is not allowed in CDSL, function calls can always be treated abstractly in this manner. The semantics $[\![f]\!]$ is a relation not a function because we permit abstract functions, that is, those functions defined outside of CDSL, to be nondeterministic. Thus, if $\gamma \vdash e \Downarrow r$ and $\gamma \vdash e \Downarrow r'$ then r is not necessarily equal to r'. This has implications for the way in which we prove equivalence, described in Section 4.3.

Typing rules for values are provided in Figure 5. A relation to match value environments to type contexts ($\gamma :\star \Gamma$) is also provided. Note that every value binding $(x = v) \in \gamma$ does not necessarily have a corresponding type judgement $(x : \tau) \in \Gamma$; this directly implies two very useful lemmas for type safety:

Lemma 1 (Context splitting respects environment matching)**.** *If* $\gamma :\star \Gamma$ *and* $\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$*, then* $\gamma :\star \Gamma_1$ *and* $\gamma :\star \Gamma_2$*.*

Lemma 2 (Context weakening respects environment matching)**.** *If* $\gamma :\star \Gamma$ *and* $\Gamma \overset{weak}{\rightsquigarrow} \Gamma'$ *then* $\gamma :\star \Gamma'$*.*

Proof. Follows directly from the fact that splitting and weakening yield contexts that are subsets of the original context. □

$$
\begin{array}{llll}
\text{Values} & v & ::= & \{\overline{\text{a: } v_?}\} \\
 & & \mid & \texttt{True} \mid \texttt{False} \mid \cdots \\
\text{Optional values} & v_? & ::= & v \mid \bullet \\
\text{Return values} & r & ::= & \overline{v} \mid \textbf{fail}\ v\ \overline{v} \\
\text{Function semantics} & [\![f]\!] & : & \overline{v} \times r
\end{array}
$$

$$\boxed{\gamma \vdash e \Downarrow r}$$

$$\frac{(\overline{x_i = v_i}) \subseteq \gamma}{\gamma \vdash \overline{x_i} \Downarrow \overline{v_i}}\textsc{Return} \qquad \frac{(x = v) \in \gamma \quad (\overline{x_i = v_i}) \subseteq \gamma}{\gamma \vdash \textbf{fail}\ x\ \overline{x_i} \Downarrow \textbf{fail}\ v\ \overline{v_i}}\textsc{Fail}$$

$$\frac{(x = \{\text{a: } \bullet, \ldots\}, y = v) \subseteq \gamma}{\gamma \vdash \textbf{put}\ x.\text{a} := y \Downarrow \{\text{a: } v, \ldots\}}\textsc{Put} \qquad \frac{(x = \{\text{a: } v, \ldots\}) \in \gamma}{\gamma \vdash \textbf{take}\ x.\text{a} \Downarrow (v, \{\text{a: } \bullet, \ldots\})}\textsc{Take}$$

$$\frac{\gamma \vdash e \Downarrow r' \qquad \gamma \vdash B; r' \Downarrow r}{\gamma \vdash \textbf{bind!}\ (vs)\ e \Rightarrow B \Downarrow r}\textsc{Bind!} \qquad \frac{\gamma \vdash e \Downarrow r' \qquad \gamma \vdash B; r' \Downarrow r}{\gamma \vdash \textbf{bind}\ e \Rightarrow B \Downarrow r}\textsc{Bind}$$

$$\frac{(x = c) \in \gamma \qquad \gamma \vdash e_c \Downarrow r}{\gamma \vdash \textbf{if}\ x\ \textbf{then}\ e_{\texttt{True}}\ \textbf{else}\ e_{\texttt{False}} \Downarrow r}\textsc{If} \qquad \frac{(\overline{x_i = v_i}) \subseteq \gamma \quad (\overline{v_i}, r) \in [\![f]\!]}{\gamma \vdash f(\overline{x_i}) \Downarrow r}\textsc{Ap.}$$

$$\boxed{\gamma \vdash B; r' \Downarrow r}$$

$$\frac{\overline{x_i = v_i}, \gamma \vdash e \Downarrow r}{\gamma \vdash \overline{x_i}.\ e; \overline{v_i} \Downarrow r}\text{OK} \qquad \frac{y = v, \overline{y_i = v_i}, \gamma \vdash e' \Downarrow r}{\gamma \vdash \overline{x_i}.\ e\ \textbf{handle}\ y\ \overline{y_i}.\ e'; \textbf{fail}\ v\ \overline{v_i} \Downarrow r}\textsc{Fail}'$$

$$\frac{\overline{x_i = v_i}, \gamma \vdash e \Downarrow r}{\gamma \vdash \overline{x_i}.\ e\ \textbf{handle}\ y\ \overline{y_i}.\ e'; \overline{v_i} \Downarrow r}\text{OK}' \qquad \frac{x = v, \overline{x_i = v_i}, \gamma \vdash e \Downarrow r}{\gamma \vdash \textbf{handle}\ x\ \overline{x_i}.\ e; \textbf{fail}\ v\ \overline{v_i} \Downarrow r}$$

Figure 4: Value Semantics

In addition, as the value semantics ignores linear types, we can show **bang** does not affect value typing:

Lemma 3 (Applying **bang** respects value typing)**.** *If $v : t^{\mathbf{1}}$ then $v : \textbf{\textit{bang}}(t)^{\infty}$.*

Proof. Observe that $x : t^{\mathbf{1}}$ implies $x : t^{\infty}$, as the value typing rules ignore linear types. As **bang**(t) simply changes all linear types $t'^{\mathbf{1}}$ to shareable types t'^{∞} inside t, **bang** can be shown to respect value typing by simple rule induction on the premise. □

In order to prove type safety, we must also make some assumptions about the semantics of functions $[\![f]\!]$. Specifically, we must assume that functions are type-safe to show that evaluation is type-safe:

Assumption 1 (Type safety of function semantics)**.** *For each function f, if $\textbf{\textit{funcTy}}(f) = \overline{\tau_i} \to T$ and, for each i, $v_i : \tau_i$, then,*

1. *There exists an r such that $(\overline{v_i}, r) \in [\![f]\!]$ and,*
2. *For all r where $(\overline{v_i}, r) \in [\![f]\!]$, $r : T$.*

Theorem 1 (Type Safety - Value Semantics)**.**
Assuming

(1) $\Gamma \vdash e : T$ *and*

(2) $\gamma :\star \Gamma$

We show:

Progress *There exists an r such that $\gamma \vdash e \Downarrow r$.*

Preservation *For any r, if $\gamma \vdash e \Downarrow r$ then $r : T$.*

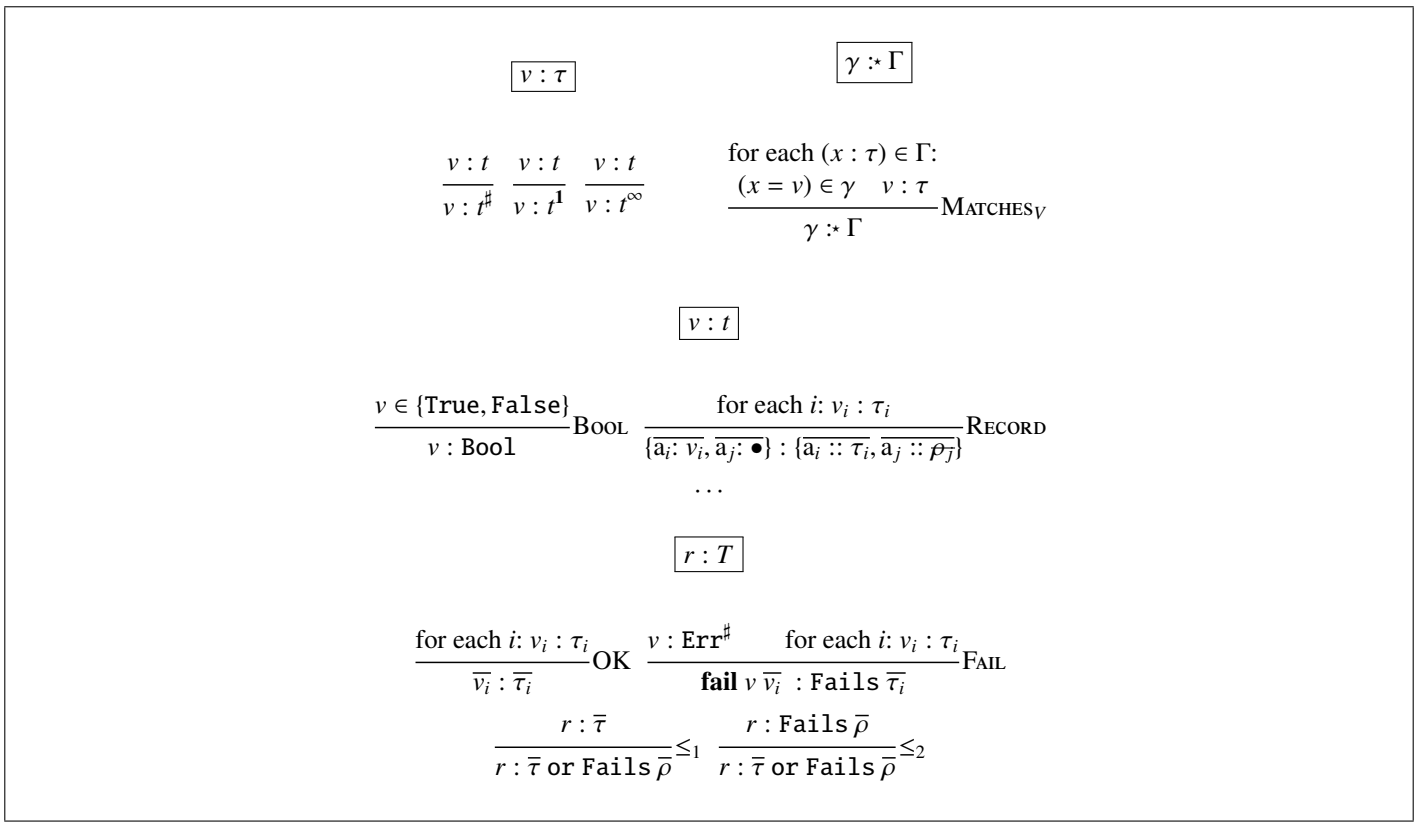

Figure 5: Value Typing Rules (Value Semantics)

Proof. By rule induction on (1), using Lemmas 1 and 2 to instantiate the induction hypothesis, Lemma 3 for the Bind! case and Assumption 1 for the case for rule Ap. □

As CDSL contains no recursion, and loops are limited to traversals of abstract data types that are known to terminate, progress in the sense proven here implies termination. Restricting CDSL to be total and terminating does not just make the formalisation simpler, it is also one of the properties we wish to ensure statically in file system code.

4.2 Update Semantics

The update semantics is provided in Figure 6. As opposed to the value semantics, the update semantics resembles that of an impure, imperative language: Values may also be pointers, locations in the mutable store σ, which is now included in the evaluation relation $\gamma \vdash \sigma; e \Downarrow! \sigma'; s$. The store itself is a map from locations to either a value or $\bullet$, which we use to indicate free space. Most of the rules are straightforward adaptations from their value-semantics equivalents, except the rules Take and Put, which now operate on pointers, and destructively update the records they point to in the store. Each function f is given an update semantics $[\![f]\!]_!$, which corresponds to its value semantics $[\![f]\!]$. A `free` function would, for example, have a value semantics which is simply a no-op, whereas the update semantics would update the store to free the value at the location specified in the argument.

The mutable store here is still quite abstract compared to the heap of bytes on an actual machine or even the typed heap of Greenaway et al. [2012]. We will bridge the remaining gap by an automatically generated refinement proof; ongoing work discussed in Section 5.

The value typing rules for the update semantics are given in Figure 7. The judgement $\sigma \vdash u : \tau \langle \mathbf{w}\text{: } L_w\ \mathbf{r}\text{: } L_r \rangle$ not only indicates that the value u under the store σ has type τ, but also that the value u contains pointers L_w and L_r, where each pointer in L_w has linear type, and is thus writable, and each pointer in L_r has shareable type, and is thus read-only. This means that we can ensure in the rule Record that all writable pointers in record values are unique – i.e, that there is no aliasing pointer, read-only or writable, to any writeable pointer. We make similar aliasing restrictions in the environment matching relation $\sigma \vdash \gamma :\star \Gamma' \langle \mathbf{w}\text{: } L_w\ \mathbf{r}\text{: } L_r \rangle$ and in the return value typing rules OK and Fail.

Note that not all the inhabited types in the value semantics are inhabited in the update semantics. For example, in the value semantics, it would make perfect sense to have an unboxed record containing a linear field. In the update semantics, unboxed values cannot contain any pointers (see rule $\mathrm{U}^\sharp$), and therefore may only contain unboxed components. Similarly, shareable values cannot contain linear components (see rule P^∞). This is essential to prevent aliasing of writeable pointers, which in turn is necessary to show the equivalence of the two semantics.

Lemma 4 (Readable pointers do not alias writable pointers)**.** *If* $\sigma \vdash u : \tau \langle \mathbf{w}\text{: } L_w\ \mathbf{r}\text{: } L_r \rangle$ *then* $L_w \cap L_r = \emptyset$.

Proof. By rule induction on the premise. □

The environment matching relation, like that of the value semantics, does not require all values in the environment to have a corresponding type judgement in the context, thus similar lemmas regarding context splitting and weakening apply, only this time, context splitting implies that the two resulting contexts do not share writeable pointers:

Lemma 5 (Context splitting respects environment matching)**.** *If* $\sigma \vdash \gamma :\star \Gamma \langle \mathbf{w}\colon L_w\ \mathbf{r}\colon L_r\rangle$ *and* $\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$*, then there exists* $L'_{w_1}, L'_{w_2} \subseteq L_w$*, and* $L'_{r_1}, L'_{r_2} \subseteq L_r$ *such that:*

1. $L'_{w_1} \cap L'_{w_2} = \emptyset$ *and*
2. $(L'_{w_1} \cup L'_{w_2}) \cap (L'_{r_1} \cup L'_{r_2}) = \emptyset$ *and*
3. $\sigma \vdash \gamma :\star \Gamma_1 \langle \mathbf{w}\colon L'_{w_1}\ \mathbf{r}\colon L'_{r_1}\rangle$ *and*
4. $\sigma \vdash \gamma :\star \Gamma_2 \langle \mathbf{w}\colon L'_{w_2}\ \mathbf{r}\colon L'_{r_2}\rangle$.

Lemma 6 (Context weakening respects environment matching)**.** *If* $\sigma \vdash \gamma :\star \Gamma \langle \mathbf{w}\colon L_w\ \mathbf{r}\colon L_r\rangle$ *and* $\Gamma \overset{weak}{\rightsquigarrow} \Gamma'$ *then there exists an* $L'_r \subseteq L_r$ *such that* $\sigma \vdash \gamma :\star \Gamma' \langle \mathbf{w}\colon L_w\ \mathbf{r}\colon L'_r\rangle$.

Proof. Similar to the proofs for Lemmas 1 and 2. □

We also prove Lemma 4, but for environment matching rather than value typing:

Lemma 7 (Readable pointers do not alias writable pointers for environments)**.** *If* $\sigma \vdash \gamma :\star \Gamma \langle \mathbf{w}\colon L_w\ \mathbf{r}\colon L_r\rangle$ *then* $L_w \cap L_r = \emptyset$.

Proof. By induction on Γ, using Lemma 4 for the inductive case. □

These non-aliasing results are crucial as they allow us to show that updates to one part of the store do not affect statements that concern another part of the store. We define a relation, called **frame** after the analogous rule from separation logic, which is a relation between an initial store, a set of input writeable pointers, a final store and a set of output writeable pointers. Written (σ, L_i) **frame** (σ', L_o), this relation denotes that, for any pointer ℓ:

Inertia If $\ell \notin L_i \cup L_o$ then $\sigma[\ell] = \sigma'[\ell]$.

Leak freedom If $\ell \in L_i$ and $\ell \notin L_o$ then $\sigma'[\ell] = \bullet$.

Fresh allocation If $\ell \in L_o$ and $\ell \notin L_i$ then $\sigma[\ell] = \bullet$.

Then, we can show that value typing, and therefore environment matching, is not affected by unrelated store updates:

Lemma 8 (Value typing framing)**.** *If*

(1) $\sigma \vdash u : \tau \langle \mathbf{w}\colon L_w\ \mathbf{r}\colon L_r\rangle$ *and*

(2) (σ, L_i) **frame** (σ', L_o) *and*

(3) $L_i \cap (L_w \cup L_r) = \emptyset$*,*

Then $\sigma' \vdash u : \tau \langle \mathbf{w}\colon L_w\ \mathbf{r}\colon L_r\rangle$

Proof. By rule induction on (1), using the definition of **frame** to show that the value u is unchanged under σ'. □

Lemma 9 (Environment matching framing)**.** *If*

(1) $\sigma \vdash \gamma :\star \Gamma \langle \mathbf{w}\colon L_w\ \mathbf{r}\colon L_r\rangle$ *and*

(2) (σ, L_i) **frame** (σ', L_o) *and*

(3) $L_i \cap (L_w \cup L_r) = \emptyset$*,*

Then $\sigma' \vdash \gamma :\star \Gamma \langle \mathbf{w}\colon L_w\ \mathbf{r}\colon L_r\rangle$

Proof. By induction on Γ, using 8 for the inductive case. □

There is also an analogous result to Lemma 3 for **bang**:

Lemma 10 (Applying **bang** respects value typing)**.** *If*
$\sigma \vdash u : t^{\mathbf{1}} \langle \mathbf{w}\colon L_w\ \mathbf{r}\colon L_r\rangle$
then $\sigma \vdash u : \boldsymbol{bang}(t)^{\infty} \langle \mathbf{w}\colon \emptyset\ \mathbf{r}\colon L_w \cup L_r\rangle$

Proof. Similar to Lemma 3. □

To prove type safety, as with the value semantics, we must assume type safety of functions that are called:

Assumption 2 (Type safety of function update semantics)**.** *For each function f, if* ***funcTy***$(f) = \overline{\tau_i} \to T$ *and, for each i, $\sigma \vdash u_i : \tau_i \langle \mathbf{w}\colon L_{w_i}\ \mathbf{r}\colon L_{r_i} \rangle$, then,*

1. *There exists σ', s such that $((\sigma, \overline{u_i}), (\sigma', s)) \in [\![f]\!]_!$ and,*
2. *For all σ', s where $((\sigma, \overline{u_i}), (\sigma', s)) \in [\![f]\!]_!$, there exists a L'_w and a $L'_r \subseteq (\bigcup_i L_{r_i})$, such that $\sigma' \vdash s : T \langle \mathbf{w}\colon L'_w\ \mathbf{r}\colon L'_r \rangle$ and $(\sigma, \bigcup_i L_{w_i})$ **frame** (σ, L'_w).*

Then, we can at last show type safety:

Theorem 2 (Type Safety - Update Semantics)**.**
Assuming

(1) $\Gamma \vdash e : T$ *and*

(2) *(2)* $\sigma \vdash \gamma :\star \Gamma \langle \mathbf{w}\colon L_w\ \mathbf{r}\colon L_r \rangle$,

We show:

Progress *There exists σ', s such that $\gamma \vdash \sigma; e \Downarrow! \sigma'; s$.*

Preservation *For any σ', s if $\gamma \vdash \sigma; e \Downarrow! \sigma'; s$, then there exists a L'_w and a $L'_r \subseteq L_r$, such that $\sigma' \vdash s : T \langle \mathbf{w}\colon L'_w\ \mathbf{r}\colon L'_r \rangle$ and (σ, L_w) **frame** (σ, L'_w).*

Proof. By rule induction on (1), using Lemmas 5 and 6 to instantiate the induction hypothesis, and Lemma 7 along with Lemma 5 for satisfying the no-alias obligations. In the case for Ap, Assumption 2 is used directly. The cases for Bind and Bind!, which have sequential composition, require Lemma 9 to properly instantiate the induction hypothesis. The case for Bind! also requires Lemma 10, and use of the type-based safety restriction to satisfy the no-alias obligations. □

This type safety result alone is sufficient to show all of the properties we wanted to ensure statically, such as termination and absence of memory leaks, however it is still cumbersome to reason about functional behaviour on the level of the update semantics. By showing the two semantics equivalent, we raise the level of abstraction for reasoning about functional correctness, so that the verifier need not be concerned with pointers or destructive update, while still having statements about the value semantics describe the behaviour of the underlying update semantics.

4.3 Equivalence

A correspondence relation between values from the update semantics (and a store) and values from the value sematics is defined in Figure 8. The relation $(\sigma, u) \sim v$ can be thought of as "flattening" all the pointers in u by looking them up in σ, giving a pointer-free result in v.

Lemma 11 (Well-typed update semantics values have a corresponding value semantics value)**.** *For any value u where $\sigma \vdash u : T \langle \mathbf{w}\colon L_w\ \mathbf{r}\colon L_r \rangle$, there exists a v such that $(\sigma, u) \sim v$.*

Proof. Rule induction on the premise. □

We also define a correspondence relation for environments, $(\sigma, \gamma_u) \overset{\Gamma}{\sim} \gamma_v$, which only compares values for which there exists a type judgement in Γ. Thus, just as with the environment matching relations above, this relation is well-behaved with respect to context splitting and weakening:

Lemma 12 (Context splitting respects environment correspondence)**.** *If $(\sigma, \gamma_u) \overset{\Gamma}{\sim} \gamma_v$ and $\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$, then $(\sigma, \gamma_u) \overset{\Gamma_1}{\sim} \gamma_v$ and $(\sigma, \gamma_u) \overset{\Gamma_2}{\sim} \gamma_v$*

Lemma 13 (Context weakening respects environment correspondence)**.** *If $(\sigma, \gamma_u) \overset{\Gamma}{\sim} \gamma_v$ and $\Gamma \overset{weak}{\rightsquigarrow} \Gamma'$, then $(\sigma, \gamma_u) \overset{\Gamma'}{\sim} \gamma_v$*

Proof. Very similar to Lemmas 1 and 2. □

We also need framing lemmas, much like environment matching:

Lemma 14 (Framing of value correspondence)**.** *If*

(1) $(\sigma, u) \sim v$ *and*

(2) $\sigma \vdash u : v \langle \mathbf{w}\colon L_w\ \mathbf{r}\colon L_r \rangle$ *and*

(3) (σ, L_i) **frame** (σ', L_o) *and*

(4) $L_i \cap (L_w \cup L_r) = \emptyset$,

Then $(\sigma', u) \sim v$.

Proof. Rule induction on (1), using (2) and (3) to show that u does not depend on any pointers updated in σ'. □

Lemma 15 (Framing of environment correspondence)**.** *If*

(1) $(\sigma, \gamma_u) \overset{\Gamma}{\sim} \gamma_v$ *and*

(2) $\sigma \vdash \gamma_u :\star \Gamma \langle \mathbf{w}\colon L_w\ \mathbf{r}\colon L_r \rangle$ *and*

(3) (σ, L_i) **frame** (σ', L_o) *and*

(4) $L_i \cap (L_w \cup L_r) = \emptyset$,

Then $(\sigma', \gamma_u) \overset{\Gamma}{\sim} \gamma_v$.

Proof. Induction on Γ, using Lemma 14 □

Furthermore, we must assume that the semantics of functions are equivalent:

Assumption 3 (Equivalence of function semantics)**.** *For each function f, where* $\textbf{\textit{funcTy}}(f) = \overline{\tau_i} \to T$ *and, for each i,* $\sigma \vdash u_i : \tau_i \langle \mathbf{w}\colon L_{w_i}\ \mathbf{r}\colon L_{r_i} \rangle$ *and* $v_i : \tau_i$ *and* $(\sigma, u_i) \sim v_i$*, then,*

1. *For all r, if* $(\overline{v_i}, r) \in [\![f]\!]$*, then there exists σ', s such that* $((\sigma, \overline{u_i}), (\sigma', s)) \in [\![f]\!]_!$ *and* $(\sigma', s) \sim r$*, and*
2. *For all σ', s, if* $((\sigma, \overline{u_i}), (\sigma', s)) \in [\![f]\!]_!$*, then there exists r such that* $(\overline{v_i}, r) \in [\![f]\!]$ *and* $(\sigma', s) \sim r$.

Now, finally, we can show equivalence:

Theorem 3 (Equivalence of Value and Update Semantics)**.**
Assuming

(1) $\Gamma \vdash e : T$ *and*

(2) $\gamma_v :\star \Gamma$ *and*

(3) $\sigma \vdash \gamma_u :\star \Gamma \langle \mathbf{w}\colon L_w\ \mathbf{r}\colon L_r \rangle$ *and*

(4) $(\sigma, \gamma_u) \overset{\Gamma}{\sim} \gamma_v$,

We show:

(U $\Longleftarrow$ V) *For all r, if* $\gamma_v \vdash e \Downarrow r$*, then there exists σ', s such that* $\gamma_u \vdash \sigma; e \Downarrow! \sigma'; s$ *and* $(\sigma', s) \sim r$.

(U $\Longrightarrow$ V) *For all σ', s, if* $\gamma_u \vdash \sigma; e \Downarrow! \sigma'; s$*, then there exists an r such that* $\gamma_v \vdash e \Downarrow r$ *and* $(\sigma', s) \sim r$.

Proof. By rule induction on (1), along very similar lines to the type safety proofs, using Theorems 1 and 2 to re-establish typing obligations, Lemmas 1, 5 and 12 to deal with context splits, Lemmas 2, 6 and 13 to deal with weakening, and Lemmas 9 and 15 to deal with store updates (in the Bind and Bind! cases). In the case for Ap, Assumption 3 is similarly used directly. □

If CDSL were fully deterministic, we could simply show that any two evaluations of the same program from corresponding environments give corresponding results, but as there are multiple possible evaluations of the same program, we must show the above two statements separately.

5 The Path to C

Our CDSL compiler already produces C code from CDSL programs. Work is still under way, however, to automatically generate proofs of correctness for this generated C code. This section gives a brief outlook on how we envision this process to complete.

The correctness statement between C and CDSL code is a refinement theorem between the CDSL program's update semantics and the semantics of the generated C code; i.e., we are generating a new proof for each program. We chose the update semantics for this proof, because it is closer to the semantics of C. With the equivalence shown in this paper, we can add another abstraction step on top of this C correctness statement. This gives us a CDSL program in value semantics that is connected by formal proof to its C implementation, and, using translation validation, to the final binary [Sewell et al., 2013]. We are also making use of the AutoCorres tool of Greenaway et al. [2012, 2014], which performs automated abstraction from C code. This means the target for our proof compiler does not need to be the low-level C semantics itself, but instead a more convenient abstraction, which simplifies automation.

In generating this proof, we exploit the purely local nature of our C code generation, which is without whole-program or interprocedural optimisations or transformations. The code generation for each individual construct therefore corresponds to precisely

one proof rule in Isabelle, which connects the CDSL semantics for that construct with its (AutoCorres-abstracted) C representation. The refinement proof for the entire program then merely needs to compose these rules appropriately.

These proof rules will depend on preconditions about the expected state of the program, for instance, about the type and validity of pointers in the heap. We plan to propagate the conditions similarly to the proof calculus of Cock et al. [2008]. Since our proof rules are specialised to CDSL and the operation of the compiler, we can predict the form of these preconditions and design proof rules to combine them. This is the basis for automating these proofs of refinement.

6 Related Work

The High-Assurance Systems Programming (HASP)[, HASP] project shares our goals of improving the reliability of systems software. It is also similar in spirit, in that they seek to make these improvements by employing formal methods as well as programming language research. HASP's systems programming language, Habit, is just like CDSL: a domain specific functional language. Providing a full formalisation of Habit's semantics to facilitate formal reasoning and verification is one of the priorities of the project. McCreight et al. [2010] show the correctness of a garbage collector in this project; however, to the best of our knowledge, there exist no full formal language semantics yet. Habit is more general than CDSL. For example, it offers support for bit level and memory based data description, which we moved to a separate domain-specific language, DDSL.

The language PacLang [Ennals et al., 2004] is a domain-specific language which uses linear types to guide optimisation of packet processing applications on network processors. The use of linear types in other areas of systems programming suggests that CDSL may find uses outside of file systems. Indeed, while PacLang is an imperative language, most PacLang programs could be translated to CDSL with little difficulty. The use of linear types in PacLang is purely designed for optimisation, not for verification, and thus its type system is much less expressive than CDSL.

Similar substructural type systems, namely uniqueness types, have been integrated in general purpose programming languages, most notably Clean [Barendsen and Smetsers, 1993]. The resulting type system is more convenient, but much more complex, with improvements on its type system proposed in de Vries et al. [2008]. In the domain for which CDSL is designed, the added convenience would buy us little, but make the proofs significantly more difficult, if at all possible.

To the best of our knowledge, Hofmann [2000] is the only work which attempts to prove the equivalence of the functional and imperative interpretation of a language with a linear type system. The paper introduces a first order functional language with linear types, not unlike CDSL, and formalises its semantics by denotation to set theory. It presents a translation of this language into C, and provides an informal proof of equivalence between the set theoretic interpretation and the C program. It is, however, not a rigorous mechanised formalisation, and the approach would be unsuitable for machine-checked verification.

The verification of file systems has received some attention, as they are a well known source of system errors. Previous manual attempts [Arkoudas et al., 2004, Damchoom and Butler, 2009, Hesselink and Lali, 2009, Schierl et al., 2009] to provide verified file systems have, however, only proven the equivalence between two and more high-level specifications. None of these efforts have managed to relate the specification to an implementation of a realistic file system. These attempts also suffered from the overwhelming size and complexity of file system implementations. In order to prove complex properties about the file system, this previous research had to introduce serious limitations resulting in oversimplified filesystems that demonstrate the verification principle, but would not be usable in practice.

7 Future Work

As mentioned in Section 2, we are currently working on implementing the proof generation, and are improving the C code generation to better correspond to the semantics defined in Isabelle. Simultaneously, we are implementing increasingly complex file systems with the help of our framework, to evaluate, improve and benchmark our approach.

Although CDSL was designed with file systems in mind, most of the observations on which we based our decisions are equally applicable to other systems code. It would be interesting to explore other domains to which our approach would be well-suited; for instance, network protocol code or marshalling/un-marshalling code in component middleware.

Our current language design is conservative in the sense that, when faced with a design trade-off between adding a feature to make CDSL more powerful on one side or preserving ease of verification and automation on the other, we usually opted for fewer features and easier verification. These decisions were informed by the case study and general experience with file system code. We left out features only as long as the language was still sufficiently expressive to get the job done.

So, for example, we opted for linear types instead of affine types, and we chose explicit error handling, even in cases where the compiler could easily infer sensible error handlers from types. These decisions are not set in stone, and should be revisited in future iterations of the language to potentially increase programmer productivity.

One problem class that is clearly important for large portions of real-life systems code is concurrency. While we have not yet addressed concurrency directly in CDSL, many of our design decisions were made with this future extension in mind.

8 Conclusion

We have presented the pure functional language CDSL, designed to express an executable specification for file systems or other, similarly structured code. The purpose of this language is to facilitate the development of fully verified systems at a substantially reduced cost.

CDSL achieves this task (1) by providing a purely functional semantic interface, which is is well suited as target for refinement proofs from a high-level specification; (2) by providing an update semantics, which facilitates the compilation to efficient C code; and (3) with a strong static semantics, guaranteeing the handling of error values, the absence of memory leaks and of errors through aliasing. Points (2) and (3) provide the basis for the automatic generation of correctness proofs for the generated code with respect to the operational semantics of CDSL.

We presented the formalisation of a core fragment of CDSL and its static semantics, as well as the dynamic value and the update semantics. Furthermore, we have formally proven the equivalence of the two dynamic semantics. This is the first fully formal proof of the equivalence of the two semantics for a language with linear types.

Acknowledgements

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

This material is based on research sponsored by Air Force Research Laboratory and the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-12-9-0179. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory, the Defense Advanced Research Projects Agency or the U.S. Government.

Store locations	ℓ		
Sets of locations	L		
Store	σ	:	$\ell \to u_?$
Values	u	::=	$\ell \mid \{\text{a: } \overline{u_?}\}$
		$\mid$	$\texttt{True} \mid \texttt{False} \mid \cdots$
Optional values	$u_?$	::=	$u \mid \bullet$
Return values	s	::=	$\overline{u} \mid \textbf{fail}\ u\ \overline{u}$
Function semantics	$[\![f]\!]_!$	:	$(\sigma \times \overline{u}) \times (s \times \sigma')$

$$\boxed{\gamma \vdash \sigma; e \Downarrow! \sigma'; s}$$

$$\frac{(\overline{x_i = u_i}) \subseteq \gamma}{\gamma \vdash \sigma; \overline{x_i} \Downarrow! \sigma; \overline{u_i}}\textsc{Return} \qquad \frac{(x = u) \in \gamma \quad (\overline{x_i = u_i}) \subseteq \gamma}{\gamma \vdash \sigma; \textbf{fail}\ x\ \overline{x_i} \Downarrow! \sigma; \textbf{fail}\ u\ \overline{u_i}}\textsc{Fail}$$

$$\frac{\begin{array}{c}(x = \ell, y = u) \subseteq \gamma \\ \sigma[\ell] = \{\text{a: } \bullet, \dots\} \\ \sigma' = \sigma[\ell := \{\text{a: } u, \dots\}]\end{array}}{\gamma \vdash \sigma; \textbf{put}\ x.\text{a} := y \Downarrow! \sigma'; \ell}\textsc{Put} \qquad \frac{\begin{array}{c}(x = \ell) \in \gamma \\ \sigma[\ell] = \{\text{a: } u, \dots\} \\ \sigma' = \sigma[\ell := \{\text{a: } \bullet, \dots\}]\end{array}}{\gamma \vdash \sigma; \textbf{take}\ x.\text{a} \Downarrow! \sigma'; (u, \ell)}\textsc{Take}$$

$$\frac{\begin{array}{c}\gamma \vdash \sigma; e \Downarrow! \sigma'; s' \\ \gamma \vdash \sigma'; B; s' \Downarrow! \sigma''; s\end{array}}{\gamma \vdash \sigma; \textbf{bind}!\ (vs)\ e \Rightarrow B \Downarrow! \sigma''; s} \qquad \frac{\begin{array}{c}\gamma \vdash \sigma; e \Downarrow! \sigma'; s' \\ \gamma \vdash \sigma'; B; s' \Downarrow! \sigma''; s\end{array}}{\gamma \vdash \sigma; \textbf{bind}\ e \Rightarrow B \Downarrow! \sigma''; s}$$

$$\frac{(x = c) \in \gamma \qquad \gamma \vdash \sigma; e_c \Downarrow! \sigma'; s}{\gamma \vdash \sigma; \textbf{if}\ x\ \textbf{then}\ e_{\texttt{True}}\ \textbf{else}\ e_{\texttt{False}} \Downarrow! \sigma'; s} \qquad \frac{\begin{array}{c}(\overline{x_i = u_i}) \subseteq \gamma \\ ((\sigma, \overline{u_i}),\ (\sigma', s)) \in [\![f]\!]_!\end{array}}{\gamma \vdash \sigma; f(\overline{x_i}) \Downarrow! \sigma'; s}$$

$$\boxed{\gamma \vdash \sigma; B; s' \Downarrow! \sigma'; s}$$

$$\frac{\overline{x_i = u_i}, \gamma \vdash \sigma; e \Downarrow! \sigma'; s}{\gamma \vdash \sigma; \overline{x_i}.\ e; \overline{u_i} \Downarrow! \sigma'; s} \qquad \frac{x = u, \overline{x_i = u_i}, \gamma \vdash \sigma; e \Downarrow! \sigma'; s}{\gamma \vdash \sigma; \textbf{handle}\ x\ \overline{x_i}.\ e; \textbf{fail}\ u\ \overline{u_i} \Downarrow! \sigma'; s}$$

$$\frac{y = u, \overline{y_i = u_i}, \gamma \vdash \sigma; e' \Downarrow! \sigma'; s}{\gamma \vdash \sigma; \overline{x_i}.\ e\ \textbf{handle}\ y\ \overline{y_i}.\ e'; \textbf{fail}\ u\ \overline{u_i} \Downarrow! \sigma'; s}\textsc{Fail}'$$

$$\frac{\overline{x_i = u_i}, \gamma \vdash \sigma; e \Downarrow! \sigma'; s}{\gamma \vdash \sigma; \overline{x_i}.\ e\ \textbf{handle}\ y\ \overline{y_i}.\ e'; \overline{u_i} \Downarrow! \sigma'; s}\textsc{OK}'$$

Figure 6: Update Semantics

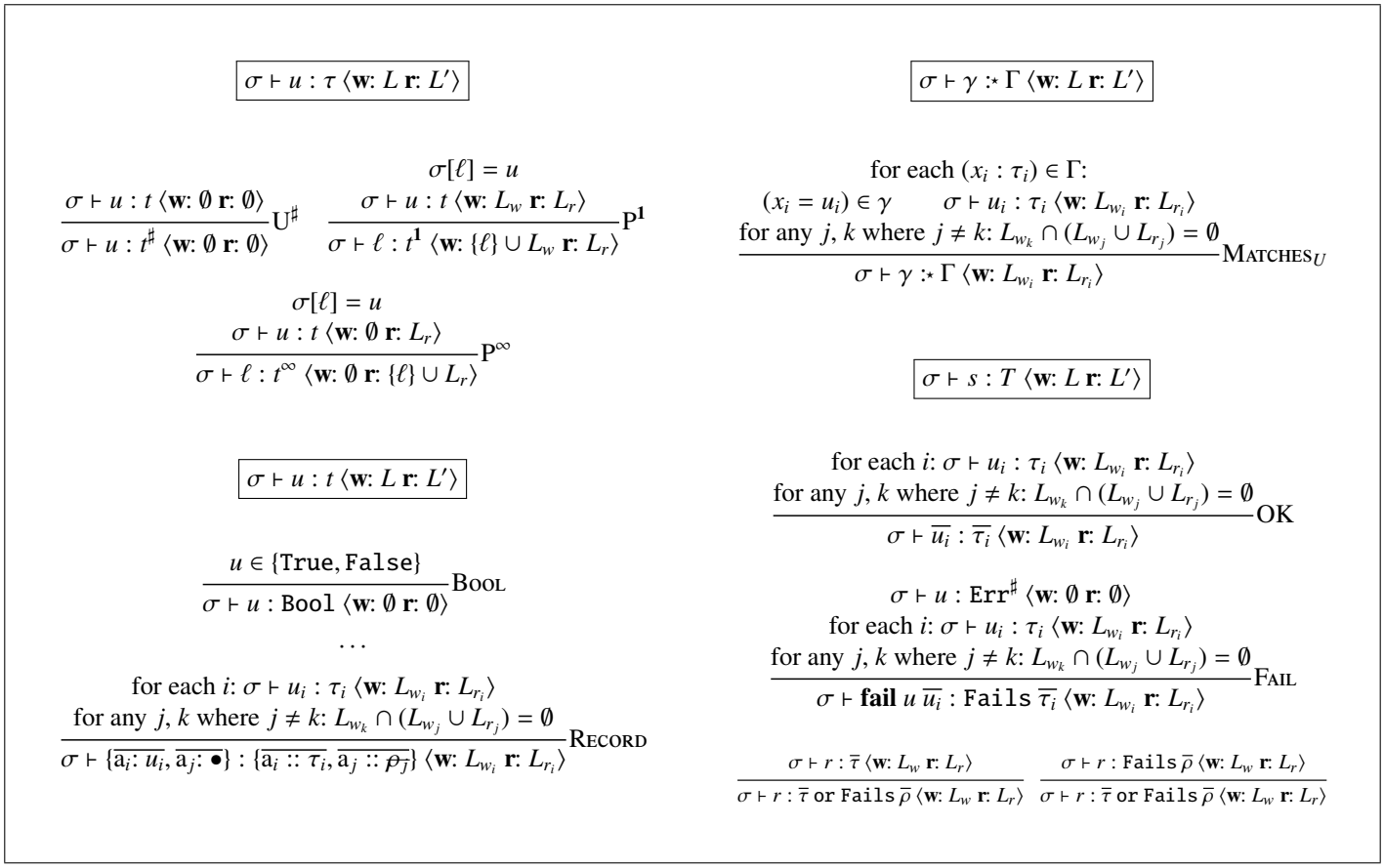

Figure 7: Value Typing Rules (Update Semantics)

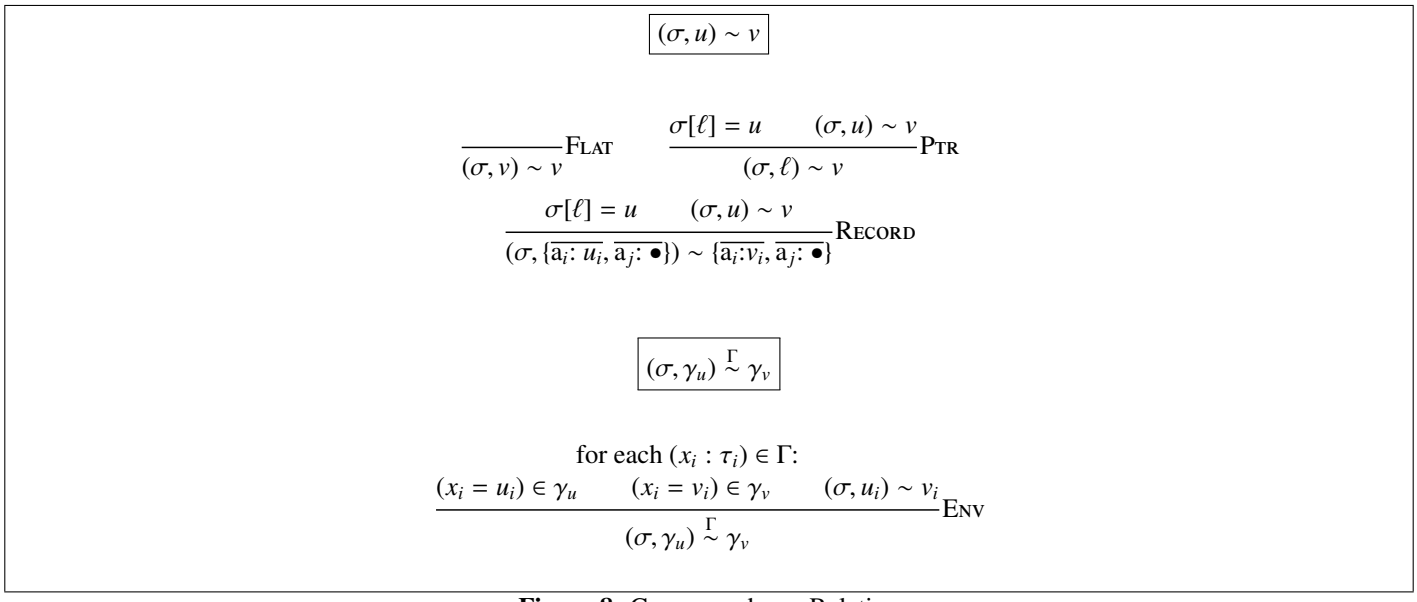

Figure 8: Correspondence Relation

Bibliography

Amal Ahmed, Matthew Fluet, and Greg Morrisett. A step-indexed model of substructural state. In 10th ICFP, pages 78–91, 2005. 7

Konstantine Arkoudas, Karen Zee, Viktor Kuncak, and Martin C. Rinard. Verifying a file system implementation. In 6thICFEM, volume 3308 of LNCS, pages 373–390, 2004. 14

Godmar Back. DataScript – a specification and scripting language for binary data. volume 2487 of LNCS, pages 66–77, 2002. 3

Thomas Ball, Ella Bounimova, Rahul Kumar, and Vladimir Levin. SLAM2: Static driver verification with under 4% false alarms. In 2010 FMCAD, pages 35–42, Lugano, Switzerland, 2010. 1

Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems. In FSTTCS, volume 761 of LNCS, pages 41–51, 1993. 14

Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. CACM, 53(2):66–75, Feb 2010. 1

David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In 21st TPHOLs, pages 167–182, Montreal, Canada, Aug 2008. doi: 10.1007/978-3-540-71067-7_16. 14

Kriangsak Damchoom and Michael J. Butler. Applying event and machine decomposition to a flash-based filestore in event-b. In SBMF, volume 5902 of LNCS, pages 134–152, 2009. 14

Nicolaas Govert de Bruijn. Lambda Calculus Notation with Nameless Dummies: a Tool for Automatic Formula Manipulation with Application to the Church-Rosser Theorem. Indagationes Mathematicae (Elsevier), 34:381–392, 1972. 8

Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. Uniqueness typing simplified. In Implementation and Application of Functional Languages, volume 5083 of LNCS, pages 201–218, 2008. 14

Rob Ennals, Richard Sharp, and Alan Mycroft. Linear types for packet processing. In 13th ESOP, volume 2986 of LNCS, pages 204–218, 2004. 14

Kathleen Fisher and David Walker. The PADS project: An overview. In Int. Conf. Database Theory, pages 11–17, 2011. 3

David Greenaway, June Andronick, and Gerwin Klein. Bridging the gap: Automatic verified abstraction of C. In 3rd ITP, volume 7406 of LNCS, pages 99–115, Princeton, New Jersey, Aug 2012. 10, 13

David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. Don't sweat the small stuff: Formal verification of C code without the pain. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 429–439, Edinburgh, UK, Jun 2014. doi: 10.1145/2594291.2594296. 13

The High Assurance Systems Programming Project (HASP). The Habit programming language: The revised preliminary report, Nov 2010. URL http://hasp.cs.pdx.edu/habit-report-Nov2010.pdf. 14

Wim H. Hesselink and Muhammad Ikram Lali. Formalizing a hierarchical file system. ENTCS, 259:67–85, 2009. 14

Martin Hofmann. A type system for bounded space and functional in-place update–extended abstract. In ESOP, volume 1782 of LNCS, pages 165–179, 2000. 14

William Jannen, Chia-Che Tsai, and Donald E. Porter. Virtualize storage, not disks. In HotOS, pages 1–7, Santa Ana Pueblo, NM, USA, May 2013. 1

Gabi Keller, Toby Murray, Sidney Amani, Liam O'Connor-Davis, Zilin Chen, Leonid Ryzhyk, Gerwin Klein, and Gernot Heiser. File systems deserve verification too! In PLOS, pages 1–7, Farmington, Pennsylvania, USA, Nov 2013. doi: 10.1145/2525528.2525530. 2, 3

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. Trans. Comp. Syst., 32(1):2:1–2:70, Feb 2014. doi: 10.1145/2560537. 1

Peter Lammich and Andreas Lochbihler. The Isabelle collections framework. In 1st ITP, volume 6172 of LNCS, pages 339–354, 2010. 3

Andreas Lochbihler. Light-weight containers for Isabelle: Efficient, extensible, nestable. In 4th ITP, volume 7998 of LNCS, pages 116–132, 2013. 3

Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. In 11th Usenix FAST, San Jose, CA, USA, Feb 2013. 1

Peter J. McCann and Satish Chandra. PacketTypes: abstract specification of network protocol messages. In SIGCOMM, pages 321–333, Stockholm, Sweden, 2000. 3

Andrew McCreight, Tim Chevalier, and Andrew Tolmach. A certified framework for compiling and executing garbage-collected languages. In 15th ICFP, pages 273–284, 2010. 14

Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with Termite. In SOSP, Big Sky, MT, USA, Oct 2009. 1

Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. SIGPLAN Lisp Pointers, V(1): 288–298, Jan 1992. 7

Andreas Schierl, Gerhard Schellhorn, Dominik Haneberg, and Wolfgang Reif. Abstract specification of the ubifs file system for flash memory. In FM, pages 190–206, 2009. 14

Thomas Sewell, Magnus Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In PLDI, pages 471–481, Seattle, Washington, USA, Jun 2013. 2, 13

Philip Wadler. Linear types can change the world! In Programming Concepts and Methods, 1990. 3, 4, 5