Manifest Contracts for Datatypes

Taro Sekiyama
Graduate School of Informatics
Kyoto University
t-sekiym@kuis.kyoto-u.ac.jp

Yuki Nishida
Graduate School of Informatics
Kyoto University
nishida@fos.kuis.kyoto-u.ac.jp

Atsushi Igarashi
Graduate School of Informatics
Kyoto University
igarashi@kuis.kyoto-u.ac.jp

Abstract

We study algebraic datatypes in a manifest contract system, a software contract system where contract information occurs as refinement types. We first compare two simple approaches: refinements on type constructors and refinements on data constructors. For example, lists of positive integers can be described by $\{l{:}\mathsf{int\ list} \mid \mathsf{for\_all}\,(\lambda y.y > 0)\,l\}$ in the former, whereas by a user-defined datatype $\mathsf{pos\_list}$ with cons of type $\{x{:}\mathsf{int} \mid x > 0\} \times \mathsf{pos\_list} \rightarrow \mathsf{pos\_list}$ in the latter. The two approaches are complementary: the former makes it easier for a programmer to write types and the latter enables more efficient contract checking. To take the best of both worlds, we propose (1) a syntactic translation from refinements on type constructors to equivalent refinements on data constructors and (2) dynamically checked casts between different but compatible datatypes such as $\mathsf{int\ list}$ and $\mathsf{pos\_list}$. We define a manifest contract calculus $\lambda^{H}_{\mathtt{dt}}$ to formalize the semantics of the casts and prove that the translation is correct.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures; D.2.4 [*Software Engineering*]: Software/Program Verification—Programming by contracts

General Terms Languages, Design, Theory

Keywords algebraic datatypes, datatype translation, contract checking, refinement types

1. Introduction

1.1 Background: Software Contracts

Software contracts are a prominent tool to develop robust software. Contracts allow programmers to write specifications in the same programming language as that used to write programs, making it possible to check such specifications at run time. They are provided as libraries or primitive constructs in various practical programming languages. For example, the C language provides the assert macro to check at run time that a given Boolean expression evaluates to true and the Eiffel language [21] provides a dedicated construct to specify and check pre- and postconditions of methods and class invariants. Racket is a representative functional language with higher-order contracts, based on the seminal work by Findler and Felleisen [9].

Although contracts were originally conceived as a mechanism to check software properties dynamically, it was also clear that contract checking could cause significant overhead for various reasons and that it would be desirable to find contract violations earlier than run time. A lot of research has been conducted to address these problems. For example, Herman, Tomb, and Flanagan [16] and Siek and Wadler [25] address the space-efficiency problem that inserting contract checking can degrade tail calls into non-tail calls; Findler, Guo, and Rogers [10] introduce lazy contract checking to address the problem that naive contract checking for datatypes can make asymptotic time complexity worse; and there is a lot of work on static analysis/verification of contracts [11, 15, 18, 22, 32, 33] to find out statically which contract checking always succeeds in order to eliminate such successful contract checking for optimization. The last line of work is also closely related to static refinement checking [17, 24, 27, 31].

In this paper, we revisit the problem of contract checking on datatypes, especially in the context of *manifest contracts* [4, 11, 13, 14, 18].

1.2 Manifest Contracts and Two Approaches to Datatypes

In a manifest contract system, unlike more traditional (dubbed *latent* by Greenberg, Pierce, and Weirich [14]) contract systems, contract information occurs as *refinement types* of the form $\{x{:}T \mid e\}$. This form of type denotes the subset of values v of type T satisfying the Boolean expression e, namely, $e\,\{v/x\}$ reduces to true. For example, $\{x{:}\mathsf{int} \mid x > 0\}$ denotes positive integers. Refinement types can be introduced by using *casts*, which involve run-time checking. A cast $\langle T \Leftarrow S\rangle^{\ell}$ means that, when applied to a value of the source type S, it is checked that the value can behave as a value of the target type T. For example, the cast application $\langle\{x{:}\mathsf{int} \mid x > 0\} \Leftarrow \mathsf{int}\rangle^{\ell}\,5$ succeeds, after confirming $5 > 0$ returns true, and returns 5. If a cast fails, an uncatchable exception will be raised with the label ℓ to identify which cast has failed. Computational calculi of manifest contracts have been studied as theoretical frameworks for hybrid contract checking [11, 18], in which contract checking is performed both statically and dynamically. The idea behind hybrid contract checking is to check, for each cast $\langle T \Leftarrow S\rangle^{\ell}$, whether it is an upcast, or equivalently, S is a subtype of T. Upcasts are proved to be contextually equivalent to the identity functions and so safe to be eliminated. The other casts are still subject to run-time check.

There are two approaches to specifying contracts for data structures. One is to put refinements on the type constructor for a plain data structure and the other is to put refinements on (types for) data constructors. For example, a type slist for sorted integer lists can be written $\{x{:}\mathsf{int\ list} \mid \mathsf{sorted}\,x\}$ in which sorted is a familiar Boolean function that returns whether the argument list is sorted

POPL '15, January 15–17, 2015, Mumbai, India.

ACM 978-1-4503-3300-9/15/01…$15.00.
http://dx.doi.org/10.1145/2676726.2676996

in the former, or defined as another datatype with refined cons of type x:int $\times$ $\{xs$:slist | nil xs or $x \leq$ head $xs\}$ → slist in the latter. Here, the argument type is a dependent product type, expressing the relationship between the two components in the pair. However, as pointed out by Findler, Guo, and Rogers [10], neither approach by itself is very satisfactory.

On the one hand, the former approach, which is arguably easier for ordinary programmers, may cause significant overhead in contract checking to make asymptotic time complexity worse. To see how it happens, let us consider function `insert_sort` for insertion sort. The sorting function and its auxiliary function insert can be defined in the ML-like syntax as follows.

```
type slist1 = {x:int list | sorted x}

let rec insert (x:int) (l:slist1) : slist1 =
  match l with
  | []     -> ⟨slist1 ⇐ int list⟩^ℓ1 [x]
  | y::ys ->
    if x <= y then ⟨slist1 ⇐ int list⟩^ℓ2 (x::l)
    else ⟨slist1 ⇐ int list⟩^ℓ3
      (y::(insert x (⟨slist1 ⇐ int list⟩^ℓ4 ys)))

let rec insert_sort (l:int list) : slist1 =
  match l with
  | []     -> []
  | x::xs -> insert x (insert_sort xs)
```

Without gray-colored casts, `insert_sort` would be an ordinary insertion-sort function. However, in `insert`, the four subexpressions `[x]`, `x::l`, `y::(insert x ys)` and `ys`, which are given type `int list`, are actually expected to have type `slist1` by the context. To fill the gap[1], we have to check whether these subexpressions satisfy the contract `sorted`. Notice that these casts cannot be eliminated by simple subtype checking because `int list` is obviously not a subtype of `slist1`. As far as we understand, existing technologies cannot verify these casts will be successful, at least, without giving hints to the verifier. Unfortunately, leaving these casts (especially ones with ℓ_2, ℓ_3, and ℓ_4) has an unpleasant effect: They traverse the entire lists to check sortedness, even though the lists have already been sorted, making the asymptotic time complexity of `insert` from $O(m)$ to $O(m^2)$, where m stands for the length of the input.

On the other hand, the latter approach, which exploits refinement in argument types of data constructors, does not have this efficiency problem (if not always). For example, we can define sorted lists as a datatype with refined constructors:

```
type slist2 =
  SNil
| SCons of
    x:int × {xs:slist2 | nil xs or x <= head xs}
```

Here, `nil` and `head` are functions[2] that return whether a given list is empty and the first element of a given list, respectively, and a type of the form $x{:}T_1 \times T_2$ is a dependent product type, which denotes pairs (v_1, v_2) of values such that v_1 and v_2 are of types T_1 and $T_2\,\{v_1/x\}$, respectively. So, `SCons` takes an integer `x` and a (sorted) list whose head (if any) is equal to or greater than `x`. Using `slist2`, we can modify the functions `insert` and `insert_sort` to perform less dynamic checking.

```
let rec insert' (x:int) (l:slist2) : slist2 =
  match l with
  | SNil -> SCons (x,⟨slist_x ⇐ slist2⟩^ℓ SNil)
  | SCons (y, ys) ->
    if x <= y then SCons (x,⟨slist_x ⇐ slist2⟩^ℓ l)
    else SCons
           (y,⟨slist_y ⇐ slist2⟩^ℓ (insert' x ys))

let rec insert_sort' (l:int list) : slist2 =
  match l with
  | []     -> SNil
  | x::xs -> insert' x (insert_sort' xs)
```

Here, $\texttt{slist}_e$ stands for `{xs:slist2 | nil xs or` e `<= head xs}`. Since the contract in the cast $\langle\texttt{slist}_\texttt{x} \Leftarrow \texttt{slist2}\rangle^\ell$ does not traverse `xs`, it is more efficient than the first definition; in fact, the time complexity of `insert'` remains to be $O(m)$. Moreover, it would be possible to eliminate the cast on `l` by collecting conditions (`l` is equal to `SCons(y, ys)` and `x <= y`) guarding this branch [24]. (It is more difficult to eliminate the other cast because the verifier would have to know that the head of the list returned by the recursive call to `insert'` is greater than `y`.)

However, this approach has complementary problems. First, we have to maintain the predicate function `sorted` and the corresponding type definition `slist2` separately. Second, it may not be a trivial task to write down the specification as data constructor refinement. For example, consider the type of lists whose elements contain a given integer n. A refinement type of such lists can be written `{l:int list | member n l}` using the familiar `member` function. One possible datatype definition corresponding to the refinement type above would be given by using an auxiliary datatype, parameterized over an integer `n` and a Boolean flag `p` to represent whether `n` has to appear in a list.

```
type incl_aux⟨p:bool, n:int⟩ =
  LNil  of {unit|not p}
| LCons of x:int × incl_aux⟨not (x=n) and p, n⟩

type list_including⟨n:int⟩ = incl_aux⟨true,n⟩
```

(Notice that `incl_aux⟨false,n⟩` is essentially `int list` and, if a list without n is given type `incl_aux`$\langle p, n\rangle$, then p must be `false`.) We do not think it is as easy to come up with a datatype definition like this as the refinement type above.

Another issue is interoperability between a plain type and its refined versions: Just as casts between `slist1` and `int list` are allowed, we would hope that the language supports casts between `slist2` and `int list`, even when they have different sets of data constructors. Such interoperability is crucial for code reuse [10]—without it, we must reimplement many list-processing functions, such as `sort`, `member`, `map`, etc., every time a refined datatype is given. As pointed out in Vazou, Rondon, and Jhala [27], one can give one generic datatype definition, which is parameterized over predicates on components of the datatype, and instantiate it to obtain plain and sorted list types but, as we will show later, refined datatype definitions may naturally come with more data constructors than the plain one, in which case parameterization would not work (the number of constructors is the same for every instantiation).

In short, the two approaches are complementary.

1.3 Our Contributions

Our work aims at taking the best of both worlds. First, we give a provably correct syntactic translation from refinements on type constructors, such as the Boolean function `sorted`, to equivalent type definitions where data constructors are refined, namely, `slist2`. This translation is closely related to the work by Atkey, Johann, and Ghani [3] and McBride [20], also concerned about

[1] Actually, there are subexpressions whose expected types are `int list` but actual types are `slist1`. We assume that `slist1` can be converted to `int list` for free.

[2] Precisely speaking, these functions have to be defined together with `slist2` but we omit them for brevity.

systematic generation of a new datatype; see Section 5 for comparison. Second, we extend casts so that casts between similar but different datatypes (what we call compatible types, which are declared explicitly in datatype definitions) are possible. For example, $\langle$`slist2` $\Leftarrow$ `int list`$\rangle^\ell$`(1 :: 2 :: [])` yields `SCons(1, SCons(2, SNil))`, whereas $\langle$`slist2` $\Leftarrow$ `int list`$\rangle^\ell$ `(1 :: 0 :: [])` raises blame ℓ. Thanks to the two ideas, a programmer can automatically derive a datatype definition from a familiar Boolean function, exploit the resulting datatype for less dynamic checking as we saw in the example of insertion sort, and also use it, when necessary, as if it were a refinement type using the Boolean function.

We formalize these ideas as a manifest contract calculus $\lambda^H_{\mathtt{dt}}$ and prove basic properties such as subject reduction and progress. We follow the existing approach, advocated by Belo et al. [4], to defining a manifest calculus without subtyping but improve it by modifying the semantics of casts slightly and simplifying the type equivalence relation. These changes play a crucial role in proving subject reduction and other semantic properties such as parametricity. We also give a first *syntactic* proof of the property that "if a program is given a refinement type $\{x{:}T \mid e\}$ and it results in a value v, then v satisfies the predicate e" in the context of manifest calculi. This property was proved by using semantic methods in the literature [14, 18]. A syntactic proof would have been possible for a polymorphic manifest calculus $\mathrm{F_H}$ [4] but the metatheory of $\mathrm{F_H}$ depends on a few conjectures, which unfortunately turned out to be false recently (personal communication).

Our contributions are summarized as follows:

- We propose casts between compatible datatypes to enhance interoperability among a plain datatype and its refined versions.
- We define a manifest contract calculus $\lambda^H_{\mathtt{dt}}$ to formalize the semantics of the casts.
- We formally define a translation from refinements on type constructors to type definitions where data constructors are refined and prove the translation is correct.

We also have a toy implementation of $\lambda^H_{\mathtt{dt}}$ on top of OCaml and Camlp4 and it is available at `http://www.fos.kuis.kyoto-u.ac.jp/~t-sekiym/papers/rech/`. A full version with proofs is also found there.

We note that this work gives type translation but does *not* give translation from a program with refinement types to one with refined datatypes, so if a programmer has a program with, for example, `slist1`, then he has to rewrite it to one with a datatype like `slist2` by hand. Automatic program transformation is left as future work.

The rest of the paper is organized as follows. Section 2 gives an overview of our datatype mechanism and Section 3 formalizes $\lambda^H_{\mathtt{dt}}$ and shows its type soundness. Then, Section 4 gives a translation from refinement types to datatypes and proves its correctness. We discuss related work in Section 5 and conclude in Section 6.

2. Overview

In this section, we informally describe our proposals of datatype definitions, casts between compatible datatypes, and translation, mainly by means of examples.

As we have seen already in the example of sorted lists, our datatype definition allows the argument types of data constructors to be refined using the set comprehension notation $\{x{:}T \mid e\}$ and dependent product types $x{:}T_1 \times T_2$. We also allow parameterization over terms as in `incl_aux` in the previous section.

2.1 Casts for Datatypes

As we have discussed in the introduction, in order to enhance interoperability between refined datatypes, we allow casts between two different datatypes if they are "compatible"; in other words, compatibility is used to disallow casts between unrelated types (for example, the integer type and a function type). Compatibility for types other than datatypes means that two types are the same by ignoring refinements; compatibility for datatypes means that there is a correspondence between the sets of the data constructors from two datatypes and the argument types of the corresponding constructors are also compatible. In our proposal, a correspondence between constructors has to be explicitly declared. So, the type `slist2` in the previous section is actually written as follows:

```
type slist2 =
  SNil  || []
| SCons || (::) of
    x:int × {xs:slist2 | nil xs or x <= head xs}
```

The symbol `||` followed by a data constructor from an existing datatype declares how constructors correspond. The types `int list` and `slist2` are compatible because both `SNil` and `[]` take no arguments and the argument type `x:int × {xs:slist2 | nil xs or x <= head xs}` of `SCons` is compatible with `int × int list` of `(::)`. (Precisely speaking, compatibility is defined coinductively.) Readers may think that explicit declaration of a correspondence of data constructors seems cumbersome. However, we could replace these declarations by a compatibility declaration for type names as `slist2 || int list` and let the system infer the correspondence between data constructors. Such inference is easy for many cases, where the argument types of data constructors are of different shapes, as in this example.

A cast for datatypes converts data constructors to the corresponding ones and puts a new cast on components. For example, $\langle \mathsf{slist} \Leftarrow \mathsf{int\ list}\rangle^\ell\,(1 :: 2 :: 3 :: [\,])$ reduces to $\mathsf{SCons}\,(1, \mathsf{SCons}\,(2, \mathsf{SCons}\,(3, \mathsf{SNil})))$ as follows:

$$
\begin{array}{l}
\quad \langle \mathsf{slist} \Leftarrow \mathsf{int\ list}\rangle^\ell\,(1 :: 2 :: 3 :: [\,]) \\
\rightarrow \mathsf{SCons}(\langle x{:}\mathsf{int} \times \{xs{:}\mathsf{slist} \mid \mathsf{nil}\ xs \text{ or } x \leq \mathsf{head}\ xs\} \Leftarrow \mathsf{int} \times \mathsf{int\ list}\rangle^\ell \\
\qquad\qquad (1, 2 :: 3 :: [\,])) \\
\rightarrow \mathsf{SCons}(1, \langle \{xs{:}\mathsf{slist} \mid \mathsf{nil}\ xs \text{ or } 1 \leq \mathsf{head}\ xs\} \Leftarrow \mathsf{int\ list}\rangle^\ell\,(2 :: 3 :: [\,])) \\
\rightarrow \ldots \\
\rightarrow \mathsf{SCons}\,(1, \mathsf{SCons}\,(2, \mathsf{SCons}\,(3, \mathsf{SNil})))
\end{array}
$$

In the example above, the correspondence between data constructors is bijective but we actually allow nonbijective correspondence, too. This means that a new datatype can have two or more (or even no) data constructors corresponding to a single data constructor from an existing type. For example, an alternative definition of `list_including` is as follows:

```
type list_including⟨n:int⟩ =
  LConsEq  || (::) of {x:int|x=n} × int list
| LConsNEq || (::) of
                {x:int|x<>n} × list_including⟨n⟩
```

This version of `list_including` has no constructors compatible to `Nil` because the empty list does not include `n`. By contrast, there are two constructors, `LConsEq` and `LConsNEq`, both compatible to `(::)`. The constructor `LConsEq` is used to construct lists where the head is equal to n, and `LConsNEq` to construct lists where the head is not equal to n but the tail list includes n. A cast to the new version of `list_including` works by choosing either `LConsEq` or `LConsNEq`, depending on the head of the input list:

$$
\begin{array}{ll}
\langle \mathsf{list\_including}\langle 0\rangle \Leftarrow \mathsf{int\ list}\rangle^\ell\,[\,] & \longrightarrow^* \quad \Uparrow\ell \\
\langle \mathsf{list\_including}\langle 0\rangle \Leftarrow \mathsf{int\ list}\rangle^\ell\,(2 :: 0 :: 1 :: [\,]) & \longrightarrow^* \\
\qquad\qquad \mathsf{LConsNEq}\,(2, (\mathsf{LConsEq}\,(0, 1 :: [\,]))) &
\end{array}
$$

This cast does not have to traverse a given list when it succeeds (notice int list in the argument type of LConsEq and $1 :: [\,]$ in the second example above).

Although it is fairly clear how to choose an appropriate constructor in the example above, it may not be as easy in general. In the formal semantics we give in this paper, we model these choices as oracles. In practice, a constructor choice function is specified along with a datatype definition either manually or often automatically—in fact, we will show that a constructor choice function can be systematically derived when a new datatype is generated from our translation. More interestingly, the asymptotic time complexity of the cast from a plain list to the generated datatype is no worse than the cast to the original refinement type. In this sense, the translation preserves efficiency of casts. This efficiency preservation lets us conjecture that, when a programmer rewrites a program with the refinement type to one with the generated datatype, the asymptotic time complexity of the latter program becomes no worse than the former. We discuss efficiency preservation in detail in Section 4.3.

Allowing nonbijective correspondence between constructors simplifies our translation and makes dynamic contract checking more efficient as in the example above.

2.2 Ideas for Translation

We informally describe the ideas behind our translation through the example of list_including above. We start with the refinement type $\{x{:}\mathsf{int\ list} \mid \mathsf{member}\ n\ x\}$, where `member n x` is a usual function, which returns whether `n` appears in list `x`:

```
let rec member (n:int) (l:int list) =
  match l with
  | []     -> false
  | x::xs ->
    if x = n then true
             else member n xs
```

Through this paper, we always suppose that some logical operations such as && and || are desugared to simplify our formalization, and so here we write `if x = n then true else member n xs` instead of `x = n || member n xs`. We examine how list_including corresponds to `member`. For reference, the definition of list_including is shown below again:

```
type list_including⟨n:int⟩ =
| LConsEq   || (::) of {x:int|x=n} × int list
| LConsNEq  || (::) of
                 {x:int|x<>n} × list_including⟨n⟩
```

We expect that a value of list_including$\langle n \rangle$ returns true when it is passed to `member n` (modulo constructor names).

It is not difficult to observe two things. First, each constructor and its argument type represent when the predicate returns true. In this example, there are two reasons that `member n x` returns true: either (1) `n` is equal to the first element of `x` or (2) `n` is not equal to the first element of `x` but `member n` is true for the tail of `x`. The constructors LConsEq and LConsNEq and their argument types represent these conditions. Since `member n x` never returns true when `x` is the empty list, there is no constructor in list_including. Second, a recursive call on a substructure corresponds to type-level recursion: `member n xs` in the `else`-branch in `member` is represented by list_including$\langle n \rangle$ in the argument type of LConsNEq.

So, the basic idea of our translation scheme is to analyze the body of a given predicate function and collect guarding conditions on branches reaching `true`. As mentioned above, recursive calls on the tail become type-level recursion. This correspondence between execution paths and data constructors is also useful to derive a constructor choice function for a cast. For example, a cast to list_including$\langle n \rangle$ will choose LConsEq when (the list being checked is not empty and) the head is equal to n, just because LConsEq corresponds to the path guarded by `x=n` in the definition of `member`.

Types

$$T \quad ::= \quad \mathsf{Bool} \mid x{:}T_1 \to T_2 \mid x{:}T_1 \times T_2 \mid \{x{:}T \mid e\} \mid \tau\langle e \rangle$$

Constants, Values, Terms

$$\begin{array}{lcl} c & ::= & \mathsf{true} \mid \mathsf{false} \\ v & ::= & c \mid \mathtt{fix}\, f(x{:}T_1){:}T_2 = e \mid \langle T_1 \Leftarrow T_2 \rangle^{\ell} \mid (v_1, v_2) \mid C\langle e \rangle v \\ e & ::= & c \mid x \mid \mathtt{fix}\, f(x{:}T_1){:}T_2 = e \mid e_1\, e_2 \mid (e_1, e_2) \mid e.1 \mid e.2 \mid \\ & & C\langle e_1 \rangle e_2 \mid \mathsf{match}\ e\ \mathsf{with}\ \overline{C_i\, x_i \to e_i}^{\,i} \mid \\ & & \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \langle T_1 \Leftarrow T_2 \rangle^{\ell} \end{array}$$

Datatype definitions

$$\begin{array}{lcl} \varsigma & ::= & \tau\,\langle x{:}T \rangle = \overline{C_i \,:\, T_i}^{\,i} \mid \tau\,\langle x{:}T \rangle = \overline{C_i \parallel D_i \,:\, T_i}^{\,i} \\ \Sigma & ::= & \varnothing \mid \Sigma, \varsigma \end{array}$$

Figure 1. Program syntax.

$\mathit{TypDefOf}_{\Sigma}(\tau)$	The definition of τ.
$\mathit{ArgTypeOf}_{\Sigma}(\tau)$	The parameter name and its type of τ.
$\mathit{CtrsOf}_{\Sigma}(\tau)$	The set of constructors that belong to τ.
$\mathit{TypSpecOf}_{\Sigma}(C)$	The type specification of C.
$\mathit{TypNameOf}_{\Sigma}(C)$	The data type that C belongs to.
$\mathit{CtrArgOf}_{\Sigma}(C)$	The argument type of C.

Table 1. Lookup functions.

3. A Manifest Contract Calculus $\lambda^{H}_{\mathtt{dt}}$

We formalize a manifest contract calculus $\lambda^{H}_{\mathtt{dt}}$ of datatypes with its syntax, type system, and operational semantics, and prove its type soundness. Following Belo et al. [4], we drop subtyping and subsumption from the core of the calculus to simplify the definition and metatheory.

In the following, we write a sequence with an overline: for example, $\overline{C_i}^{\,i \in \{1,\ldots,n\}}$ means a sequence $C_1, \ldots, C_n$ of data constructors. We often omit the index set $\{1, \ldots, n\}$ when it is clear from the context or not important. Given a binary relation R, the relation R^* denotes the reflexive and transitive closure of R.

3.1 Syntax

We present the program syntax of $\lambda^{H}_{\mathtt{dt}}$ in Figure 1, where there are various metavariables: T ranges over types, τ names of datatypes, C and D constructors, c constants, x, y, z, f, etc. variables, v values, e terms, ℓ blame labels, Γ typing contexts, ς datatype definitions, Σ type definition environments.

Types consist of base types (we have only Boolean here but addition of other base types causes no problems), dependent function types, dependent product types, refinement types, and datatypes. In a dependent function type $x{:}T_1 \to T_2$ and a dependent product type $x{:}T_1 \times T_2$, variable x is bound in T_2. A refinement type $\{x{:}T \mid e\}$, in which x is bound in e, denotes the subset of type T whose value v satisfies the Boolean contract e, that is, $e\,\{v/x\}$ evaluates to true. Finally, a datatype $\tau\langle e \rangle$ takes the form of an application of τ to a term e.

Note that, unlike some refinement type systems [17, 24, 27, 28, 30, 31], which aim at decidable static verification, the predicate e is allowed to be an arbitrary Boolean expression, which may diverge or raise blame. As we will see soon, however, no computation is involved with typing rules for source programs and it is easy to show decidability of typing for source programs. In fact, two types with different predicates such as $\{x{:}\mathsf{int} \mid x > 2\}$ and $\{x{:}\mathsf{int} \mid x > 1 + 1\}$ are always distinguished and a cast is required to convert from one type to the other. Static verification amounts to checking a given cast is in fact an upcast, where the source type is a subtype of the target, and subtyping is not, in general, decidable but the language is not equipped with subsumption.

Terms are basically those from the λ-calculus with Booleans, recursive functions, products, datatypes, and casts. A term $\mathtt{fix}\, f(x{:}T_1){:}T_2 = e$ represents a recursive function in which variables x and f denote an argument and the function itself, respectively, and are bound in e. We often omit type annotations. A data constructor application $C\langle e_1 \rangle e_2$ takes two arguments: e_1 represents one for the type definition and e_2 for data constructors, respectively. A match expression $\mathsf{match}\, e \,\mathsf{with}\, \overline{C_i\, x_i \to e_i}^{\,i}$ is as usual and binds each variable x_i in e_i.

The last form is a cast $\langle T_1 \Leftarrow T_2 \rangle^{\ell}$, consisting of a target type T_1, a source type T_2, and a label ℓ, and, when applied to a value v of type T_2, checks that the value v can behave as T_1. The label ℓ is used to identify the cast when it is blamed.

A datatype definition ς can take two forms. The form $\tau\,\langle x{:}T \rangle = \overline{C_i : T_i}^{\,i}$, where x is bound in $\overline{T_i}^{\,i}$, declares a datatype τ, parameterized over x of type T, with data constructors C_i whose argument types are T_i. The other form $\tau\,\langle x{:}T \rangle = \overline{C_i \parallel D_i : T_i}^{\,i}$ is the same except that it declares that C_i is compatible with D_i from another datatype.

A type definition environment Σ is a sequence of datatype definitions. We assume that datatype and constructor names declared in a type definition environment are distinct. Table 1 shows metafunctions to look up information on datatype definitions. Their definitions are omitted since they are straightforward. A type specification, returned by *TypSpecOf* and written $x{:}T_1 \rightarrowtail T_2 \rightarrowtail \tau\langle x \rangle$, of a constructor C consists of the datatype τ that C belongs to, the parameter x of τ and the type T_1 of x, and the argument type T_2 of C. In other words, $\tau = \mathit{TypNameOf}_{\Sigma}(C)$, $x{:}T_1 = \mathit{ArgTypeOf}_{\Sigma}(\tau)$ and $T_2 = \mathit{CtrArgOf}_{\Sigma}(C)$. We omit the subscript Σ from these metafunctions for brevity if it is clear from the context.

We use the following familiar notations. We write $\mathsf{FV}(e)$ to denote the set of free variables in a term e, and $e\,\{e'/x\}$ capture avoiding substitution of e' for x in e. We apply similar notations to values and types. We say that a term/value/type is closed if it has no free variables, and identify α-equivalent ones. In addition, we introduce several shorthands. A function type $T_1 \to T_2$ means $x{:}T_1 \to T_2$ where the variable x does not occur free in T_2. We write $\lambda\, x{:}T_1.e$ to denote $\mathtt{fix}\, f(x{:}T_1){:}T_2 = e$ if f does not occur in the term e. A let-expression $\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2$ denotes $(\lambda\, x{:}T.e_2)\; e_1$ where T is an appropriate type. Finally, a datatype τ is said to be *monomorphic* if the definition of τ does not refer to a type argument variable, and then we abbreviate $\tau\langle e \rangle$ to τ.

3.2 Type System

This section introduces a type system for source programs in $\lambda^{H}_{\mathtt{dt}}$; later we extend the syntax to include run-time terms to define operational semantics and give additional typing rules for those terms. The type system consists of three judgments: context well-formedness $\Sigma \vdash \Gamma$, type well-formedness $\Sigma; \Gamma \vdash T$, and typing $\Sigma; \Gamma \vdash e : T$. Here, a typing context Γ is a sequence of variable declarations:

$$\Gamma ::= \varnothing \mid \Gamma, x{:}T$$

where declared variables are pairwise distinct. We show inference rules in Figure 2, where a type definition environment Σ in judgments are omitted for simplification. Typing rules for atomic terms, such as Booleans, variables, etc. demand that types of a typing context of a judgment be well-formed; in other rules, well-formedness of a typing context and a type of a term is shown as a derived property.

Inference rules for context and type well-formedness judgments are standard except for WT_DATATYPE, which requires an argument to a datatype τ to be typed at the declared argument type.

Most of typing rules are also standard or similar to the previous work [4]. The rule T_CAST means that the source and target types in a cast have to be compatible. Intuitively, two types are compatible when a cast from one type to the other may succeed. More formally, type compatibility, written $T_1 \parallel T_2$, is the least congruence satisfying rules in the bottom of Figure 2: the rule C_REFINEL allows casts from and to refinement types; and the rule C_DATATYPE says that if datatypes are declared to be compatible in the type definition, then they are compatible. The typing rule T_CTR demands that arguments e_2 and e_1 respect the argument types of C and the datatype that C belongs to, respectively. The rule T_MATCH for match expressions demands the matched term e_0 to be typed at a datatype $\tau\langle e \rangle$. Using the metafunction *CtrsOf*, the rule demands that the patterns $\overline{C_i\, x_i}^{\,i}$ be exhaustive. Moreover, each branch e_i has to be given the same type T, which cannot contain pattern variables x_i (and so is well formed under Γ).

3.3 Semantics

The semantics of $\lambda^{H}_{\mathtt{dt}}$ is given in the small-step style by using two relations over closed terms: the reduction relation ($\rightsquigarrow$), which represents basic computation such as β-reduction, and the evaluation relation ($\longrightarrow$), in which a subexpression is reduced.

The semantics is parameterized by a type definition environment and a *constructor choice function* δ, which is a partial function that maps a term of the form $\langle \tau_1\langle e_1 \rangle \Leftarrow \tau_2\langle e_2 \rangle \rangle^{\ell}\; C_2\langle e \rangle v$ to a constructor C_1. We introduce this function as an oracle to decide which constructor a given constructor is converted to by a cast between datatypes, as discussed in Section 2. The constructor C_1 has to not only belong to τ_1 but also be compatible with C_2. We will give a more precise condition on δ later.

Precisely speaking, the two relations are parameterized by Σ and δ but we fix certain Σ and δ in what follows and usually omit them from relations and judgments.

Before reduction and evaluation rules, we introduce several run-time terms to express dynamic contract checking in the semantics. These run-time terms are assumed not to appear in a source program (or datatype definitions). The syntax is extended as below:

$$e \quad ::= \quad \ldots \mid \Uparrow\!\ell \mid \langle \{x{:}T \mid e_1\}, e_2, v \rangle^{\ell} \mid \langle\!\langle \{x{:}T \mid e_1\}, e_2 \rangle\!\rangle^{\ell}$$

The term $\Uparrow\!\ell$ denotes a cast failure blaming ℓ, which identifies which cast failed. An active check $\langle \{x{:}T \mid e_1\}, e_2, v \rangle^{\ell}$ verifies that the value v of type T satisfies the contract e_1. The term e_2 represents an intermediate state of a check, which starts by reducing $e_1\,\{v/x\}$. If the check succeeds, namely e_2 reduces to true, then the active check evaluates to v; otherwise, if e_2 reduces to false, then it is blamed with ℓ. A waiting check $\langle\!\langle \{x{:}T \mid e_1\}, e_2 \rangle\!\rangle^{\ell}$, which appears when an application of a cast to a refinement type is reduced, checks that the value of e_2 satisfies e_1. Waiting checks are introduced to avoid a technical problem recently found in Belo et al. [4]. We will discuss it in more detail at the end of this section.

Figure 3 shows reduction and evaluation rules. Reduction rules are standard except for those about casts and active/waiting checks. There are six reduction rules for casts. The rule R_BASE means that a cast between the same base type simply behaves like an identity function. The rule R_FUN, which shows that casts between function types behave like function contracts [6, 14], produces a lambda abstraction which wraps the value v with the contravariant cast $\langle T_{21} \Leftarrow T_{11} \rangle^{\ell}$ between the argument types and the covariant cast $\langle T_{12} \Leftarrow T_{22} \rangle^{\ell}$ between the return types. To avoid capture of the bound variable of T_{12}, we take a fresh variable y and rename

$\boxed{\vdash \Gamma}$ **Typing Context Well-Formedness Rules**

$$\frac{}{\vdash \varnothing}\ \textsc{WC\_Empty} \qquad \frac{\vdash \Gamma \quad \Gamma \vdash T}{\vdash \Gamma, x{:}T}\ \textsc{WC\_ExtendVar}$$

$\boxed{\Gamma \vdash T}$ **Type Well-Formedness Rules**

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Bool}}\ \textsc{WT\_Base} \qquad \frac{\Gamma \vdash T_1 \quad \Gamma, x{:}T_1 \vdash T_2}{\Gamma \vdash x : T_1 \to T_2}\ \textsc{WT\_Fun} \qquad \frac{\Gamma \vdash T_1 \quad \Gamma, x{:}T_1 \vdash T_2}{\Gamma \vdash x{:}T_1 \times T_2}\ \textsc{WT\_Prod}$$

$$\frac{\Gamma \vdash T \quad \Gamma, x{:}T \vdash e : \mathsf{Bool}}{\Gamma \vdash \{x{:}T \mid e\}}\ \textsc{WT\_Refine} \qquad \frac{\mathit{ArgTypeOf}(\tau) = x{:}T \quad \Gamma \vdash e : T}{\Gamma \vdash \tau\langle e\rangle}\ \textsc{WT\_Datatype}$$

$\boxed{\Gamma \vdash e : T}$ **Typing Rules**

$$\frac{\vdash \Gamma \quad c \in \{\mathsf{true}, \mathsf{false}\}}{\Gamma \vdash c : \mathsf{Bool}}\ \textsc{T\_Const} \qquad \frac{\vdash \Gamma \quad x{:}T \in \Gamma}{\Gamma \vdash x : T}\ \textsc{T\_Var} \qquad \frac{\Gamma, f{:}(x{:}T_1 \to T_2), x{:}T_1 \vdash e : T_2 \quad f \notin \mathsf{FV}(T_2)}{\Gamma \vdash \mathtt{fix}\, f(x{:}T_1){:}T_2 = e : x{:}T_1 \to T_2}\ \textsc{T\_Abs}$$

$$\frac{\Gamma \vdash T_1 \quad \Gamma \vdash T_2 \quad T_1 \parallel T_2}{\Gamma \vdash \langle T_1 \Leftarrow T_2\rangle^{\ell} : T_2 \to T_1}\ \textsc{T\_Cast} \qquad \frac{\Gamma \vdash e_1 : x{:}T_1 \to T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\, e_2 : T_2\,\{e_2/x\}}\ \textsc{T\_App}$$

$$\frac{\Gamma, x{:}T_1 \vdash T_2 \quad \Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2\,\{e_1/x\}}{\Gamma \vdash (e_1, e_2) : x{:}T_1 \times T_2}\ \textsc{T\_Pair} \qquad \frac{\Gamma \vdash e : x{:}T_1 \times T_2}{\Gamma \vdash e.1 : T_1}\ \textsc{T\_Proj1} \qquad \frac{\Gamma \vdash e : x{:}T_1 \times T_2}{\Gamma \vdash e.2 : T_2\,\{e.1/x\}}\ \textsc{T\_Proj2}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T}\ \textsc{T\_If} \qquad \frac{\begin{array}{c}\mathit{TypSpecOf}(C) = x{:}T_1 \rightarrowtail T_2 \rightarrowtail \tau\langle x\rangle \\ \Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2\,\{e_1/x\} \quad \Gamma \vdash \tau\langle e_1\rangle\end{array}}{\Gamma \vdash C\langle e_1\rangle e_2 : \tau\langle e_1\rangle}\ \textsc{T\_Ctr}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : \tau\langle e\rangle \quad \Gamma \vdash T \quad \mathit{CtrsOf}(\tau) = \overline{C_i}^{\,i \in \{1,\ldots,n\}} \quad \mathit{ArgTypeOf}(\tau) = y{:}T' \\ \text{for all } i,\ \mathit{CtrArgOf}(C_i) = T_i \quad \text{for all } i,\ \Gamma, x_i{:}T_i\,\{e/y\} \vdash e_i : T\end{array}}{\Gamma \vdash \mathsf{match}\ e_0\ \mathsf{with}\ \overline{C_i\, x_i \to e_i}^{\,i \in \{1,\ldots,n\}} : T}\ \textsc{T\_Match}$$

$\boxed{T_1 \parallel T_2}$ **Type Compatibility**

$$\frac{T_1 \parallel T_2}{\{x{:}T_1 \mid e_1\} \parallel T_2}\ \textsc{C\_RefineL} \qquad \frac{\mathit{TypDefOf}(\tau_1) = (\mathsf{type}\, \tau_1\, \langle x{:}T\rangle = \overline{C_i \parallel D_i : T_i}^{\,i}) \quad \text{for all } i,\ \mathit{TypNameOf}(D_i) = \tau_2}{\tau_1\langle e_1\rangle \parallel \tau_2\langle e_2\rangle}\ \textsc{C\_Datatype}$$

Figure 2. Type system.

variable x in T_{22} to it. Similar renaming is performed in R_PROD. The rule R_PROD means that elements v_1 and v_2 are checked by covariant casts obtained by decomposing the source and target types. The rules R_FORGET and R_PRECHECK are applied when source and target types of a cast are refinement types, respectively: the rule R_FORGET peels the outermost refinement of the source type; and the rule R_PRECHECK means that inner refinements in the target type are first checked and then the outermost one is. The side condition in R_PRECHECK are needed to make the semantics deterministic. For example, the term $\langle\{x{:}\mathsf{int} \mid 0 < x + 1\} \Leftarrow \{x{:}\mathsf{int} \mid 0 < x\}\rangle^{\ell}\, v$ reduces to $\langle\langle\{x{:}\mathsf{int} \mid 0 < x + 1\}, \langle\mathsf{int} \Leftarrow \mathsf{int}\rangle^{\ell}\, v\rangle\rangle^{\ell}$ by applying first R_FORGET and then R_PRECHECK. A waiting check turns into an active check when its second argument becomes a value (R_CHECK).

There are three rules R_DATATYPE, R_DATATYPEMONO, and R_DATATYPEFAIL for datatype casts. The rule R_DATATYPE is applied when the choice function δ gives the constructor C_1; then the original constructor argument v is passed to a cast between the argument types of C_2 and C_1. Here, note that e_1 and e_2 are substituted for variables x_1 and x_2 in the argument types of C_1 and C_2, respectively, because these types depend on these variables. The rule R_DATATYPEMONO is similar to R_BASE. The rule R_DATATYPEFAIL says that, if the choice function δ is undefined for the cast, the cast application is blamed with ℓ.

The last three rules R_CHECK, R_OK, and R_FAIL follow the intuitive meaning of active checks explained above.

Evaluation rules are also shown in Figure 3. Here, evaluation contexts [8], ranged over by E, are defined as usual:

$$\begin{array}{l} E ::= [\,] \mid E\, e_2 \mid v_1\, E \mid (E, e_2) \mid (v_1, E) \mid E.1 \mid E.2 \mid C\langle e_1\rangle E \mid \\ \quad \mathsf{match}\ E\ \mathsf{with}\ \overline{C_i\, x_i \to e_i}^{\,i} \mid \mathsf{if}\ E\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \\ \quad \langle\{x{:}T \mid e\}, E, v\rangle^{\ell} \mid \langle\langle\{x{:}T \mid e\}, E\rangle\rangle^{\ell} \end{array}$$

The rule E_RED means that evaluation proceeds by reducing the redex indicated by an evaluation context; the rule E_BLAME means that a raised blame will abort program execution.

3.4 Type Soundness

We show type soundness of $\lambda^{H}_{\mathtt{dt}}$. As usual, type soundness means that a well-typed term does not go wrong and is proved via subject reduction and progress [23, 29]. Moreover, we will show that, if a term is given a refinement type, its value (if it exists) satisfies the contract. This last property, which was proved by using semantic methods [4, 18], is proved purely syntactically for the first time.

Before stating the type soundness theorem, we start with extending the type system to run-time terms, and define well-formedness of type definition environments and constructor choice functions.

3.4.1 Typing for Run-time Terms

Typing rules for run-time terms are shown in Figure 4. The rule T_BLAME means that a blame $\Uparrow\ell$ can have any type because it is an exception. In the rule T_ACHECK for an active check $\langle\{x{:}T \mid e_1\}, e_2, v\rangle^{\ell}$, the last premise $e_1\,\{v/x\} \longrightarrow^{*} e_2$ means that e_2 is an intermediate state of checking, which started from

$\boxed{e_1 \rightsquigarrow e_2}$ **Reduction Rules**

$$
\begin{array}{lll}
(\mathtt{fix}\, f(x{:}T_1){:}T_2 = e)\, v & \rightsquigarrow & e\,\{v/x, \mathtt{fix}\, f(x{:}T_1){:}T_2 = e/f\} \quad \text{R\_BETA} \\
(v_1, v_2).1 \rightsquigarrow v_1 \quad \text{R\_PROJ1} & & \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \rightsquigarrow e_1 \quad \text{R\_IFTRUE} \\
(v_1, v_2).2 \rightsquigarrow v_2 \quad \text{R\_PROJ2} & & \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 \rightsquigarrow e_2 \quad \text{R\_IFFALSE} \\
\mathsf{match}\ C_j\langle e\rangle v\ \mathsf{with}\ \overline{C_i\, x_i \to e_i}^{\,i} & \rightsquigarrow & e_j\,\{v/x_j\} \quad (\text{where } C_j \in \overline{C_i}^{\,i}) \quad \text{R\_MATCH} \\
\langle \mathsf{Bool} \Leftarrow \mathsf{Bool}\rangle^{\ell}\, v & \rightsquigarrow & v \quad \text{R\_BASE} \\
\langle x{:}T_{11} \to T_{12} \Leftarrow x{:}T_{21} \to T_{22}\rangle^{\ell}\, v & \rightsquigarrow & (\lambda x{:}T_{11}.\mathsf{let}\ y = \langle T_{21} \Leftarrow T_{11}\rangle^{\ell}\, x\ \mathsf{in}\ \langle T_{12} \Leftarrow (T_{22}\,\{y/x\})\rangle^{\ell}\,(v\, y)) \\
& & (\text{where } y \text{ is fresh}) \quad \text{R\_FUN} \\
\langle x{:}T_{11} \times T_{12} \Leftarrow x{:}T_{21} \times T_{22}\rangle^{\ell}\,(v_1, v_2) & \rightsquigarrow & \mathsf{let}\ x = \langle T_{11} \Leftarrow T_{21}\rangle^{\ell}\, v_1\ \mathsf{in}\ (x, \langle T_{12} \Leftarrow (T_{22}\,\{v_1/x\})\rangle^{\ell}\, v_2) \quad \text{R\_PROD} \\
\langle T_1 \Leftarrow \{x{:}T_2 \mid e\}\rangle^{\ell}\, v & \rightsquigarrow & \langle T_1 \Leftarrow T_2\rangle^{\ell}\, v \quad \text{R\_FORGET} \\
\langle \{x{:}T_1 \mid e\} \Leftarrow T_2\rangle^{\ell}\, v & \rightsquigarrow & \langle\!\langle \{x{:}T_1 \mid e\}, \langle T_1 \Leftarrow T_2\rangle^{\ell}\, v\rangle\!\rangle^{\ell} \quad \text{R\_PRECHECK} \\
& & (\text{where } T_2 \text{ is not a refinement type}) \\
\langle \tau_1\langle e_1\rangle \Leftarrow \tau_2\langle e_2\rangle\rangle^{\ell}\, C_2\langle e\rangle v & \rightsquigarrow & C_1\langle e_1\rangle(\langle T_1'\,\{e_1/x_1\} \Leftarrow T_2'\,\{e_2/x_2\}\rangle^{\ell}\, v) \quad \text{R\_DATATYPE} \\
\multicolumn{3}{r}{(\text{where } \tau_1 \neq \tau_2 \text{ or } \tau_1 \text{ is not monomorphic, and } \delta(\langle \tau_1\langle e_1\rangle \Leftarrow \tau_2\langle e_2\rangle\rangle^{\ell}\, C_2\langle e\rangle v) = C_1 \text{ and}} \\
\multicolumn{3}{r}{\mathit{ArgTypeOf}(\tau_i) = x_i{:}T_i \text{ and } \mathit{CtrArgOf}(C_i) = T_i' \text{ for } i \in \{1,2\})} \\
\langle \tau \Leftarrow \tau\rangle^{\ell}\, v & \rightsquigarrow & v \quad \text{R\_DATATYPEMONO} \\
\langle \tau_1\langle e_1\rangle \Leftarrow \tau_2\langle e_2\rangle\rangle^{\ell}\, v & \rightsquigarrow & \Uparrow\ell \quad \text{R\_DATATYPEFAIL} \\
\multicolumn{3}{r}{(\text{where } \tau_1 \neq \tau_2 \text{ or } \tau_1 \text{ is not monomorphic, and } \delta(\langle \tau_1\langle e_1\rangle \Leftarrow \tau_2\langle e_2\rangle\rangle^{\ell}\, v) \text{ is undefined})} \\
\langle\!\langle \{x{:}T \mid e\}, v\rangle\!\rangle^{\ell} & \rightsquigarrow & \langle \{x{:}T \mid e\}, e\,\{v/x\}, v\rangle^{\ell} \quad \text{R\_CHECK} \\
\langle \{x{:}T \mid e\}, \mathsf{true}, v\rangle^{\ell} \rightsquigarrow v \quad \text{R\_OK} & & \langle \{x{:}T \mid e\}, \mathsf{false}, v\rangle^{\ell} \rightsquigarrow \Uparrow\ell \quad \text{R\_FAIL}
\end{array}
$$

$\boxed{e_1 \longrightarrow e_2}$ **Evaluation Rules**

$$
\frac{e_1 \rightsquigarrow e_2}{E[e_1] \longrightarrow E[e_2]}\ \text{E\_RED} \qquad\qquad \frac{E \neq [\,]}{E[\Uparrow\ell] \longrightarrow \Uparrow\ell}\ \text{E\_BLAME}
$$

Figure 3. Semantics.

$\boxed{\Gamma \vdash e : T}$

$$
\frac{\vdash \Gamma \quad \varnothing \vdash T}{\Gamma \vdash \Uparrow\ell : T}\ \text{T\_BLAME} \qquad
\frac{\vdash \Gamma \quad \varnothing \vdash \{x{:}T \mid e_1\} \quad \varnothing \vdash v : T \quad \varnothing \vdash e_2 : \mathsf{Bool} \quad e_1\,\{v/x\} \longrightarrow^{*} e_2}{\Gamma \vdash \langle \{x{:}T \mid e_1\}, e_2, v\rangle^{\ell} : \{x{:}T \mid e_1\}}\ \text{T\_ACHECK} \qquad
\frac{\vdash \Gamma \quad \varnothing \vdash \{x{:}T \mid e_1\} \quad \varnothing \vdash e_2 : T}{\Gamma \vdash \langle\!\langle \{x{:}T \mid e_1\}, e_2\rangle\!\rangle^{\ell} : \{x{:}T \mid e_1\}}\ \text{T\_WCHECK}
$$

$$
\frac{\vdash \Gamma \quad \varnothing \vdash e : T_1 \quad T_1 \equiv T_2 \quad \varnothing \vdash T_2}{\Gamma \vdash e : T_2}\ \text{T\_CONV} \qquad
\frac{\vdash \Gamma \quad \varnothing \vdash v : \{x{:}T \mid e\}}{\Gamma \vdash v : T}\ \text{T\_FORGET} \qquad
\frac{\vdash \Gamma \quad \varnothing \vdash \{x{:}T \mid e\} \quad \varnothing \vdash v : T \quad e\,\{v/x\} \longrightarrow^{*} \mathsf{true}}{\Gamma \vdash v : \{x{:}T \mid e\}}\ \text{T\_EXACT}
$$

Figure 4. Typing rules for run-time terms.

$e_1\,\{v_1/x\}$. This reference to the semantics in the typing rule is unusual but is important in Belo et al.'s syntactic approach. The rule T_WCHECK for a waiting check is easy to understand. The rule T_FORGET is needed because R_FORGET peels off the refinements in the source type of a cast. The rule T_EXACT allows a value which succeeds in dynamic checking to be typed at a refinement type.

Finally, T_CONV—the heart of the approach by Belo et al.—is introduced as a technical device to prove subject reduction. To see why this rule is required, let us consider an application term $v_1\, e_2$. From T_APP, the type of $v_1\, e_2$ is $T_2\,\{e_2/x\}$ for some T_2 and x. If e_2 reduces to e_2', then the type of the application term changes to $T_2\,\{e_2'/x\}$. Since $T_2\,\{e_2/x\} \neq T_2\,\{e_2'/x\}$ in general, subject reduction would not hold. The rule T_CONV bridges the gap by allowing a term to be typed at another, but "equivalent" type. The type equivalence (denoted by $\equiv$) is given as follows.

Definition 1 (Type Equivalence)**.**

1. *The* common subexpression reduction relation $\Rrightarrow$ *over types is defined as follows:* $T_1 \Rrightarrow T_2$ *iff there exist some* T*,* x*,* e_1 *and* e_2 *such that* $T_1 = T\,\{e_1/x\}$ *and* $T_2 = T\,\{e_2/x\}$ *and* $e_1 \longrightarrow e_2$*.*
2. *The* type equivalence $\equiv$ *is the symmetric and transitive closure of* $\Rrightarrow$*.*

The type equivalence given here relates fewer terms than that by Belo et al., but is sufficient to prove subject reduction.

The fact that typing contexts in premises are empty reflects that run-time terms are closed; however, they can appear under binders as part of types (notice term substitution in the typing rules) and so weakening is needed.

3.4.2 Well-formed Type Definition Environments

Intuitively, a type definition environment is well formed when the parameter type is well formed, constructor argument types are well formed, and the argument types of compatible constructors are compatible.

Definition 2 (Well-Formed Type Definition Environments)**.**

1. *Let* $\varsigma = \tau\langle x{:}T\rangle = \overline{C_i : T_i}^{i\in\{1,\dots,n\}}$. *A type definition* ς *is* well formed under a type definition environment Σ *if it satisfies the followings: (a)* $0 < n$. *(b)* $\Sigma;\varnothing \vdash T$ *holds. (c) For any* $i \in \{1,\dots,n\}$, $\Sigma, \varsigma; x{:}T \vdash T_i$ *holds.*
2. *Let* $\varsigma = \tau\langle x{:}T\rangle = \overline{C_i \parallel D_i : T_i}^{i\in\{1,\dots,n\}}$. *A type definition* ς *is* well formed under a type definition environment Σ *if it satisfies the followings: (a)* $0 < n$. *(b)* $\Sigma;\varnothing \vdash T$ *holds. (c) For any* $i \in \{1,\dots,n\}$, $\Sigma, \varsigma; x{:}T \vdash T_i$ *holds. (d) There exists some datatype* τ' *in* Σ *such that constructors* $\overline{D_i}^{i\in\{1,\dots,n\}}$ *belong to it. (e) For any* $i \in \{1,\dots,n\}$, T_i *is compatible with the argument type of* D_i *under* Σ, ς*, that is,* $\Sigma, \varsigma \vdash T_i \parallel \mathit{CtrArgOf}_{\Sigma}(D_i)$ *holds.*
3. *A type definition environment* Σ *is* well formed *if for any* Σ_1, ς, *and* Σ_2, $\Sigma = \Sigma_1, \varsigma, \Sigma_2$ *implies that* ς *is well formed under* Σ_1. *We write* $\vdash \Sigma$ *to denote that* Σ *is well formed.*

Intuitively, a constructor choice function is well formed when it returns a constructor related by $\parallel$ in type definitions and respects term equivalence, which is defined similarly to type equivalence.

Definition 3 (Compatible Constructors)**.** *The compatibility relation* $\parallel$ *over constructors is the least equivalence relation satisfying the following rule.*

$$\frac{\mathit{TypNameOf}(C_i) = \tau \qquad \mathit{TypDefOf}(\tau) = \mathsf{type}\,\tau\langle y{:}T\rangle = \overline{C_j \parallel D_j : T_j}^{j}}{C_i \parallel D_i}$$

The function CompatCtrsOf, which maps a datatype τ *and a constructor* C *to the set of compatible constructors of* τ*, is defined as follows:*

$$\mathit{CompatCtrsOf}(\tau, C) = \{D \mid C \parallel D \text{ and } \mathit{TypNameOf}(D) = \tau\}.$$

Definition 4 (Term Equivalence)**.**

1. *The* common subexpression reduction relation $\Rrightarrow$ *over terms is defined as follows:* $e_1 \Rrightarrow e_2$ *iff there exist some* e, x, e_1' *and* e_2' *such that* $e_1 = e\{e_1'/x\}$ *and* $e_2 = e\{e_2'/x\}$ *and* $e_1' \longrightarrow e_2'$.
2. *The* term equivalence $\equiv$ *is the symmetric and transitive closure of* $\Rrightarrow$.

Definition 5 (Well-Formed Constructor Choice Functions)**.** *A constructor choice function* δ *is* well formed *iff*

1. *if* $C_1 = \delta(\langle \tau_1\langle e_1\rangle \Leftarrow \tau_2\langle e_2\rangle\rangle^{\ell}\, C_2\langle e\rangle v)$*, then* $C_1 \in \mathit{CompatCtrsOf}(\tau_1, C_2)$*; and*
2. *for any* e_1, e_2, *and* C, *if* $e_1 \equiv e_2$ *and* $\delta(e_1) = C$, *then* $\delta(e_2) = C$.

We suppose that the type definition environment and the choice function are well formed in what follows.

Lemma 1 (Subject Reduction)**.** *If* $\varnothing \vdash e : T$ *and* $e \longrightarrow e'$*, then* $\varnothing \vdash e' : T$.

Lemma 2 (Progress)**.** *If* $\varnothing \vdash e : T$*, then one of the followings holds: (1)* $e \longrightarrow e'$ *for some* e'*; (2)* e *is a value; or (3)* $e = \Uparrow\ell$ *for some* ℓ.

To show the additional property mentioned above about refinement types, we need to show that the term equivalence respects the semantics in the following sense.

Lemma 3 (Cotermination)**.** *Suppose* $e_1 \Rrightarrow^{*} e_2$.

1. *If* $e_1 \longrightarrow^{*} v_1$*, then there exist some* v_2 *such that* $e_2 \longrightarrow^{*} v_2$ *and* $v_1 \Rrightarrow^{*} v_2$.
2. *If* $e_2 \longrightarrow^{*} v_2$*, then there exist some* v_1 *such that* $e_1 \longrightarrow^{*} v_1$ *and* $v_1 \Rrightarrow^{*} v_2$.

Theorem 1 (Type Soundness)**.** *If* $\varnothing \vdash e : T$*, then one of the followings holds: (1) there exists* v *such that* $e \longrightarrow^{*} v$ *and* $\varnothing \vdash v : T$*; (2)* $e \longrightarrow^{*} \Uparrow\ell$ *for some* ℓ*; or (3) there is an infinite sequence of evaluation* $e \longrightarrow e_1 \longrightarrow \cdots$*. Moreover, if* T *is a refinement type* $\{x{:}T_0 \mid e_0\}$ *and (1) holds, then* $e_0\{v/x\} \longrightarrow^{*} \mathsf{true}$.

Proof. (1)–(3) follow from subject reduction and progress. For the additional property, it suffices to show that if $\varnothing \vdash v : T$, then v satisfies all contracts of type T. We proceed by induction on the derivation of $\varnothing \vdash v : T$. In the case of T_CONV, we use Lemma 3 and the fact that for any v', if $v' \Rrightarrow^{*} \mathsf{true}$ or $\mathsf{true} \Rrightarrow^{*} v'$, then $v' = \mathsf{true}$. □

3.4.3 Remark on Semantics of Casts

As we have mentioned, our semantics of casts for nested refinement types is slightly different from the one for F_H [4] in the following respects. First, they had a rule to remove reflexive casts

$$\langle T \Leftarrow T\rangle^{\ell}\, v \longrightarrow v$$

(for *any* type, including function and refinement types) and a rule to start an active check:

$$\langle\{x{:}T \mid e\} \Leftarrow T\rangle^{\ell}\, v \longrightarrow \langle\{x{:}T \mid e\}, e\{v/x\}, v\rangle^{\ell}$$

Second, they define type equivalence $\equiv$ based on parallel reduction. They also left the cotermination property as a conjecture. However, with these rules, cotermination does not quite hold. Consider $e = \langle\{x{:}\mathsf{Bool} \mid y\} \Leftarrow \{x{:}\mathsf{Bool} \mid \mathsf{false}\}\rangle^{\ell}\, v$. Then,

$$\begin{aligned}
&e\{\mathsf{false}/y\} \equiv e\{1 = 0/y\} \text{ and}\\
&e\{\mathsf{false}/y\} \longrightarrow v \text{ (by removing the reflexive cast), but}\\
&e\{1 = 0/y\} \longrightarrow^{*} \langle\{x{:}\mathsf{Bool} \mid 1 = 0\}, 1 = 0, v\rangle^{\ell} \longrightarrow^{*} \Uparrow\ell,
\end{aligned}$$

which is a counterexample to cotermination.[3] It is easy to construct a similar counterexample using the second rule above. This is quite bad because the cotermination property is quite important to show semantic properties such as (3) of Theorem 1 or parametricity for F_H.

The problem seems to stem from the fact that reduction of a subterm (in this case $1 = 0$) can change the cast rule to be applied. Our calculus $\lambda^{H}_{\mathbf{dt}}$ is carefully designed to avoid this problem by restricting reflexive casts (R_BASE and R_DATATYPEMONO) and introducing waiting checks. The price we pay is that we have to *prove* that reflexive casts can be eliminated.

4. Translation from Refinement Types to Datatypes

We give a translation from refinement types to datatypes and prove that the datatype obtained by the translation has the same meaning as the refinement type in the sense that a cast from the refinement type to the datatype always succeeds, and vice versa. We formalize our translation and prove its correctness using integer lists for simplicity and conciseness but our translation scheme can be generalized to other datatypes. We will informally discuss a more general case of binary trees later.

In this section, we assume that we have unit and int as base types and int list with [] and infix cons $x :: y$ as constructors. For simplicity, we also assume that the input predicate function is well typed and of the following form:

$$\mathtt{fix}\, f(y{:}T, x{:}\mathsf{int\ list}) = \mathsf{match}\, x\, \mathsf{with}\, [\,] \to e_1 \mid z_1 :: z_2 \to e_2$$

[3] Note that replacing $\equiv$ based on parallel reduction with one based on common subexpression reduction would not help. We should also note that subject reduction and progress of F_H still hold because they do not depend on cotermination.

$$
\begin{array}{lll}
\mathit{GenContracts}\,(\mathsf{true}) = \{(\mathit{None}, \mathsf{true})\} & & \mathit{GenContracts}\,(\mathsf{false}) = \varnothing \\
\mathit{GenContracts}\,(\mathsf{if}\ f\ e_1\ z_2\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3) & = & \{(\mathit{Some}\ e_1, e_2)\} \cup \\
 & & \{(e_{\mathbf{opt}}, \mathsf{if}\ f\ e_1\ z_2\ \mathsf{then\ false\ else}\ e_3') \mid (e_{\mathbf{opt}}, e_3') \in \mathit{GenContracts}\,(e_3)\} \\
 & & \hfill (\text{if } \mathsf{FV}\,(e_1) \subseteq \{y, z_1\}) \\
\mathit{GenContracts}\,(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3) & = & \{(e_{\mathbf{opt}}, \mathsf{if}\ e_1\ \mathsf{then}\ e_2'\ \mathsf{else\ false}) \mid (e_{\mathbf{opt}}, e_2') \in \mathit{GenContracts}\,(e_2)\} \cup \\
 & & \{(e_{\mathbf{opt}}, \mathsf{if}\ e_1\ \mathsf{then\ false\ else}\ e_3') \mid (e_{\mathbf{opt}}, e_3') \in \mathit{GenContracts}\,(e_3)\} \\
 & & \hfill (\text{if a term of the form } f\ e\ z_2 \text{ occurs in } e_2 \text{ or } e_3) \\
\mathit{GenContracts}\,(\mathsf{match}\ e_0\ \mathsf{with}\ \overline{C_i\ x_i \to e_i}^{i \in \{1,\ldots,n\}}) & = & \bigcup_{j \in \{1,\ldots,n\}} \{(e_{\mathbf{opt}}, \mathsf{match}\ e_0\ \mathsf{with}\ \overline{C_i\ x_i \to e_i'}^{i \in \{1,\ldots,n\}}) \mid \\
 & & \quad (e_{\mathbf{opt}}, e_j') \in \mathit{GenContracts}\,(e_j) \wedge \forall i \neq j\,.\ e_i' = \mathsf{false}\} \\
 & & \hfill (\text{if a term of the form } f\ e\ z_2 \text{ occurs in some } e_i) \\
\mathit{GenContracts}\,(e) & = & \{(\mathit{None}, e)\} \hfill (\text{otherwise})
\end{array}
$$

Figure 6. Generation of base contracts and arguments to recursive calls.

Trans
input:
$\mathtt{fix}\, f\,(y{:}T, x{:}\mathsf{int\ list}) = \mathsf{match}\ x\ \mathsf{with}\ [\,] \to e_1 \mid z_1 :: z_2 \to e_2$
returns:
1 **let** τ be a fresh type name **in**
2 **let** $\{T_i\}_i =$
$$\left\{ z_1{:}\mathsf{int} \times \{z_2{:}T_0 \mid e_0\} \;\middle|\; \begin{array}{l} (e_{\mathbf{opt}}, e) \in \mathit{GenContracts}\,(e_2), \\ (T_0, e_0) = \mathit{Aux}(\tau, e_{\mathbf{opt}}, e) \end{array} \right\} \textbf{ in}$$
3 **let** D and $\overline{D_i}^i$ be fresh constructor names, and
z be a fresh variable **in**
4 $\mathsf{type}\ \tau\,\langle y{:}T\rangle = D \parallel [\,] : \{z{:}\mathsf{unit} \mid e_1\} \mid \overline{D_i \parallel (::) : T_i}^i$

where
$\mathit{Aux}(\tau, e_{\mathbf{opt}}, e) =$
let $e' = e\,\{\mathtt{fix}\, f\,(y{:}T, x{:}\mathsf{int\ list}) = \ldots / f\}$ **in**
match $e_{\mathbf{opt}}$ **with**
$\mid \mathit{Some}\ e'' \to (\tau\langle e''\rangle, \mathsf{let}\ z_2 = \langle \mathsf{int\ list} \Leftarrow \tau\langle e''\rangle\rangle^{\ell}\ z_2\ \mathsf{in}\ e')$
$\mid \mathit{None} \to (\mathsf{int\ list}, e')$

Figure 5. Translation.

where $x \notin \mathsf{FV}\,(e_1) \cup \mathsf{FV}\,(e_2)$. We will use `sorted` as a running example. For expository reasons, the definition is slightly verbose; the nested `if` expression at the end is essentially `z1 <= y && sorted () z2`.

```
let rec sorted (y:unit, x:int list) =
  match x with
  | []     -> true
  | z1::z2 -> e2^sorted
```

where
$e_2^{\mathsf{sorted}} =$

```
match z2 with
| []     -> true
| y::ys -> if z1 <= y then
             if sorted () z2 then true
                else false
           else false
```

4.1 Translation, Formally

We show the translation function *Trans* in Figure 5 and the auxiliary function *GenContracts* in Figure 6. The main function *Trans* takes a recursive function as an input and returns a corresponding datatype definition (on line 4).

On line 2, information on how e_2, which is the contract for $(::)$, can evaluate to true is gathered by the auxiliary function *GenContracts*. In the definition, variables f, y, z_1, and z_2 come from the input function and are fixed names. This function takes an expression as an input and returns a set of pairs $(e_{\mathbf{opt}}, e_2')$, each of which expresses one execution path that returns true in e_2. e_2' is derived from e_2 by substituting false for all but one path and $e_{\mathbf{opt}}$ is the first argument to a recursive call (if any) on this path. Intuitively, conjunction of e_2' and $f\ e_{\mathbf{opt}}\ z_2$ gives one sufficient condition for e_2 to be true and disjunction of the pairs in the returned set is logically equivalent to e_2. For example, $\mathit{GenContracts}(e_2^{\mathsf{sorted}})$ returns a set consisting of (None, e_{21}) where e_{21} is

```
match z2 with [] -> true | y::ys -> false
```

and $(\mathit{Some}\ (), e_{22})$ where e_{22} is

```
match z2 with
  []     -> false
| y::ys -> if z1 <= y then true else false .
```

(Gray bits show differences from e_2^{sorted}.) The first expression means that a (non-empty) list `x` is sorted when the tail is empty; and the second means that `x` is sorted when the head `z1` is equal to or smaller than the second element `y` and the recursive call `sorted () z2` returns true. *GenContracts* performs a kind of disjunctive normal form translation and each disjunct will correspond to a data constructor in the generated datatype.

Now let us take a look at the definition of *GenContracts*. The first two clauses are trivial: if the expression is true, then it returns the trivial contract and if it is false, then this branch should not be taken and hence the empty sequence is returned. The third clause deals with a conditional on a recursive call $f\ e_1\ z_2$ on the tail. In this case, it returns $\mathit{Some}\ e_1$, to signal there is a designated recursive call in this branch, with the additional condition e_2 and also the condition when the recursive call returns false but e_3 is true. The following two clauses are for the other cases where the input expression is case analysis. In this case, from each branch, *GenContracts* recursively collects execution paths and reconstruct conditional expressions by replacing other branches with false. The side conditions on these clauses mean that we can stop DNF translation if there is no recursive calls on the tail and simply return the given contract as it is, by calling for the last clause, which deals with other forms of expressions.

The collected execution path information is further transformed into dependent product types with the help of another auxiliary function *Aux*. This function takes a pair $(e_{\mathbf{opt}}, e)$ (obtained by *GenContracts*) together with the new datatype name τ as an input and returns the base type and its refinement for the tail part. If there was no recursive call on the tail in a given execution path (namely, $e_{\mathbf{opt}} = \mathit{None}$), then the base type is $\mathsf{int\ list}$ and the refinement is e', obtained from e by replacing other recursive occurrences of f with the function itself. Otherwise, the base type is the new datatype applied to the first argument e'' to the recursive call; the refinement is essentially e' (except a cast back to int list). For example, for `sorted`, we obtain

$$T_1 = z_1{:}\mathsf{int} \times \{z_2{:}\mathsf{int\ list} \mid e_{21}\}$$

from $(None, e_{21})$ and

$$T_2 = z_1{:}\mathsf{int} \times \{z_2{:}\tau \mid \mathsf{let}\ z_2 = \langle \mathsf{int\ list} \Leftarrow \tau\rangle^{\ell}\ z_2\ \mathsf{in}\ e_{22}\}$$

from $(Some\,(), e_{22})$. T_1 is a type for singleton lists, which are trivially sorted and T_2 is a type for a list where the head is equal to or less than the second element and the tail is of type τ, which is supposed to represent sorted lists.

Finally, we combine T_i to make a complete datatype definition. The translation of `sorted` will be:

```
type sorted_t =
  SNil    || []   of {z:unit|true}
| SCons1  || (::) of z1:int × {z2:int list|e21}
| SCons2  || (::) of
    z1:int ×
    {z2:sorted_t|
     let z2 = ⟨int list ⇐ sorted_t⟩^ℓ z2 in e22}
```

Although the datatype `sorted_t` certainly represents sorted lists, its type definition is different from `slist2` given in Section 1. The difference comes from the fact that the case for (::) has a case analysis, one of whose branch has a recursive call. While it is possible to "merge" the argument types for `SCons1` and `SCons2` to make a two-constructor datatype, it is difficult in general. It is interesting future work, however, to investigate how to generate type definitions closer to programmers' expectation.

4.2 Correctness

We prove that the translation is correct in the sense that the cast from a refinement type to the datatype obtained by the translation always succeeds and vice versa. We use a metavariable F to denote the recursive function used to define the refinement type in the typing judgment and the evaluation relation. We write $\langle\Sigma, \delta\rangle; \Gamma \vdash e : T$ and $\langle\Sigma, \delta\rangle \vdash e_1 \longrightarrow e_2$ to make a type definition environment Σ and a constructor choice function δ explicit in the typing judgment and the evaluation relation.

First of all, the translation *Trans* always generates a well-formed datatype definition:

Lemma 4 (Translation Generates Well-Formed Datatype)**.** *Let* Σ *be a well-formed type definition environment,* $\Sigma; \varnothing \vdash F : T \to \mathsf{int\ list} \to \mathsf{Bool}$*. Then, the type definition* $Trans\,(F)$ *is well formed under* Σ*.*

The next theorem states that a cast from a refinement type to the generated datatype always succeeds.

Theorem 2 (From Refinement Types to Datatypes)**.** *Let* Σ *be a well-formed type definition environment,* $\Sigma; \varnothing \vdash F : T \to \mathsf{int\ list} \to \mathsf{Bool}$*,* τ *be the name of the datatype* $Trans\,(F)$*. Then, there exists some computable well-formed choice function* δ *such that, for any* $e = \langle \tau\langle e_0\rangle \Leftarrow \{x{:}\mathsf{int\ list} \mid F\ e_0\ x\}\rangle^{\ell}\ v$*, if* $(\Sigma, Trans\,(F)); \varnothing \vdash e : \tau\langle e_0\rangle$*, then* $\langle(\Sigma, Trans\,(F)), \delta\rangle \vdash e \longrightarrow^{*} v'$ *for some* v'*.*

It is a bit trickier to prove the converse because the first argument to a predicate function is always evaluated whereas a parameter to a datatype is not. So, the converse holds under the following termination condition on a datatype.

Definition 6 (Termination)**.** *Let* Σ *be a type definition environment and* δ *be a constructor choice function. A closed term* e terminates at a value under Σ and δ*, written as* $\langle\Sigma, \delta\rangle \vdash e\downarrow$*, if* $\langle\Sigma, \delta\rangle \vdash e \longrightarrow^{*} v$ *for some* v*. We say that* argument terms to datatype τ in v terminate at values under Σ and δ*, written as* $\langle\Sigma, \delta\rangle \vdash v\downarrow_{\tau}$*, if, for any* E*,* $C \in CtrsOf(\tau)$*,* e_1 *and* v_2*,* $v = E[C\langle e_1\rangle v_2]$ *implies* $\langle\Sigma, \delta\rangle \vdash e_1\downarrow$*.*

Theorem 3 (From Datatypes to Refinement Types)**.** *Let* Σ *be a well-formed type definition environment,* $\Sigma; \varnothing \vdash F : T \to \mathsf{int\ list} \to \mathsf{Bool}$*,* τ *be the name of the datatype* $Trans\,(F)$*. Then, there exists some computable well-formed choice function* δ *such that, for any* $e = \langle\{x{:}\mathsf{int\ list} \mid F\ e_0\ x\} \Leftarrow \tau\langle e_0\rangle\rangle^{\ell}\ v$*, if* $\langle(\Sigma, Trans\,(F)), \delta\rangle; \varnothing \vdash e : \{x{:}\mathsf{int\ list} \mid F\ e\ x\}$ *and* $\langle(\Sigma, Trans\,(F)), \delta\rangle \vdash v\downarrow_{\tau}$*, then* e *terminates at a value under* $\langle(\Sigma, Trans\,(F)), \delta\rangle$ *and* δ*.*

We expect that the termination condition would not be needed if we change the semantics to evaluate argument terms to datatypes.

4.3 Efficiency Preservation

In addition to correctness of the translation, we are also concerned with the following question "When I rewrite my program so that it uses the generated datatype, is it as efficient as the original one?" To answer this question positively, we consider the asymptotic time complexity of a cast *for successful inputs* (which we simply call the complexity of a cast), and show that the complexity of a cast from int list to its refinement is the same as that of a cast from int list to the datatype obtained from its refinement. Here, we consider only successful inputs because we are mainly interested in programs (or program runs) that do not raise blame, where checks caused by casts are successful.[4]

This efficiency preservation is obtained from Theorem 2 and the following observation. As stated in Theorem 2, we can construct a *computable* choice function. In fact, the algorithm of the choice function can be read off from the proof of Theorem 2: it returns constructors of the generated datatype from the execution trace of the refinement checking. Moreover, the orders of both the execution time of the algorithm and the size of output constructors from the algorithm are linear in the size of the input execution trace, which is proportional to the execution time of the refinement checking. Thus, the asymptotic time complexities of computation of the constructors and constructor replacement are no worse than that of the refinement checking.

From this observation, we can implement the cast from int list to the generated datatype by (1) checking the refinement (given to the translation) and (2) the constructor generation and replacement described above. Since the complexity of the second step is the same as that of the refinement checking, the complexities of the cast from int list to a refinement type and the generated datatype are the same.

4.4 Extension: Binary Trees

We informally describe how to extend the translation algorithm for lists to trees, a kind of data structure with a data constructor which has more than one recursive part. Here, we take binary trees as an example and show how to obtain a datatype for binary search trees from a predicate function. Although this section deals with only binary trees, this extension can be adapted for other data structures.

A datatype for binary trees and a recursive predicate function which returns whether an argument binary tree is a binary search tree or not are defined as follows:

```
type bt = L | N of int * t * t

let rec bst (min,max:int*int) (t:bt) =
 match t with
 | L          -> true
```

[4] We conjecture that, for inputs that lead to blame, the time complexity is also preserved by the translation but a proof is left for future work.

```
  | N (x,l,r) -> min<=x and x<=max and
                 bst (min,x) l and bst (x,max) r
```

Let τ be a name for the new datatype.

The translation algorithm first calls *GenContracts* with the second branch of `bst`. Observing the predicate function `bst`, we find that the body calls `bst` itself recursively for different recursive parts (`l` and `r`) with different auxiliary arguments (`(min,x)` and `(x,max)`). Thus, *GenContracts* for binary trees looks for the first argument to each recursive call, unlike *GenContracts* for lists, which stops searching for a recursive call after finding one recursive call. For our running example, taking the branch for constructor `N`, *GenContracts* for binary trees returns the singleton set $\{(\mathit{Some}\,(\mathit{min}, x), \mathit{Some}\,(x, \mathit{max}), \mathit{min} \le x \text{ and } x \le \mathit{max})\}$ (where we use the operator and instead of if expression for brevity), of which the first two optional terms denote arguments for left and right subtrees, respectively.

Next, for each element in the output from *GenContracts*, a dependent product type will be built. In this case, we obtain $T = x{:}\mathsf{int} \times l{:}\tau\langle(\mathit{min}, x)\rangle \times \{r{:}\tau\langle(x, \mathit{max})\rangle \mid \mathit{min} \le x \text{ and } x \le \mathit{max}\}$. As we have seen for lists, casts from $\tau\langle e\rangle$ back to bt may have to be inserted.

Finally, the translation makes a datatype definition by using these type arguments and the contract. For `bst`, the corresponding datatype is given as follows:

```
type t ⟨min:int,max:int⟩ =
  | SL
  | SN of x:int × l:t⟨min,x⟩ ×
            {r:t⟨x,max⟩|min<=x and x<=max}
```

4.5 Discussion

The translation algorithm works "well" for list-processing functions, in the sense that there is no reference to the input predicate function in the generated datatype, if their definitions meet the two requirements: (1) recursive calls are given the tail part of the input list and occur linearly for each execution path; and (2) free variables in arguments to recursive calls are only the argument variable y and the head variable z_1. Specifically, translation works as we expect if given functions are written in the `fold_left` form, or more generally, in the primitive recursion form where the result of a recursive call is used at most once for each execution path. In contrast, there can remain recursive calls to an input predicate function in the generated datatype when the predicate function does not meet these requirements. This happens when there is a recursive call on lists other than the tail of the input or, as in the following (admittedly quite artificial) example, when recursive calls which return `true` occur twice or more in one execution path:

```
let rec f () (l:int list) =
  match x with
  | []     -> true
  | x::xs -> f () xs and f () xs
```

or when e_2 includes non-branching constructs as in

```
let rec f (y:int) (x:int list) =
  match x with
  | []      -> true
  | z1::z2 -> let z = 5 + y in f z z2
```

In these cases, generation of a datatype itself succeeds but the obtained datatype is probably not what we expect because `f` is not eliminated.

Although our translation works well for many predicates, there is a lot of room to improve. First, the current translation algorithm could generate a datatype with too many constructors even if some of them can be "merged". For example, we demonstrated that the translation generated a datatype with three constructors from predicate function `sorted`, but we can give a datatype with only two constructors for it as shown in Section 2.1. Second, our translation algorithm works only for a single recursive Boolean function and so we cannot obtain a datatype from other forms of refinements, for example, conjunction of two predicate function calls. This also means that the translation cannot deal with a predicate function that returns additional information by using, say, an option type.

Our translation assumes an input refinement to be of a certain form. We think, however, that it is not so restrictive, because we can transform refinements before applying our method. For example, a predicate function of the form

$$\text{if } e_1 \text{ then } (\text{match } x \text{ with } [\,] \to e_{11} \mid z_1 :: z_2 \to e_{12}) \text{ else } e_2$$

can be transformed to

$$\begin{aligned} \text{match } x \text{ with } [\,] \quad &\to \text{if } e_1 \text{ then } e_{11} \text{ else } e_2 \mid \\ z_1 :: z_2 &\to \text{if } e_1 \text{ then } e_{12} \text{ else } e_2. \end{aligned}$$

Even if such transformation cannot be applied, we can always insert pattern matching on the input list in the beginning of a predicate refinement. (It may be the case, though, that we do not obtain an expected type definition.)

5. Related Work

Contracts for datatypes. There has been much work about lambda calculi with higher-order contracts since the seminal work by Findler and Felleisen [9], but little of them considers algebraic datatypes in detail and compare the two approaches to datatypes with contracts. In particular, as far as we know there is no work on conversion between compatible datatypes. One notable exception is Findler et al. [10], who compare the two approaches to datatypes and introduce lazy contract checking in an eager language. Lazy contract checking delays contract checking for arguments to data constructor until they are used. As they already point out, one drawback of lazy contract checking is that it is not suitable for checking, where relationship between elements in a data structure is important. For example, if we take the head of an arbitrary list, which is subject to sortedness checking, it simply returns its head discarding the tail without verifying the tail is sorted. Chitil [7] also made a similar observation in the work on lazy contracts in a lazy language.

Knowles et al. [19] developed Sage, a programming language based on a manifest contract calculus with first-class types and dynamic type. Sage can deal with datatypes with refined constructors by Church-encoding, but does not formalize them in its core calculus. In particular, Knowles et al. did not clarify how casts between datatypes work. Dminor [5], proposed by Bierman et al., is a first-order functional programming language with refinement types, type-test and semantic subtyping. The combination of these features is as powerful as various types such as algebraic datatypes, intersection types, and union types can be encoded. Unlike our calculus, Dminor does not deal with higher-order functions and dynamic checking with type conversion.

Xu [32] developed a hybrid contract checker for OCaml. In the static checking phase, the checker performs symbolic simplification of program components wrapped by contracts, with the help of context information, to remove blames. If a blame remains in the simplified programs, the compiler reports errors, or it issues warnings and leaves contract checking to run time. Although the checker supports variant types (i.e., datatypes where constructors have no refinements), it does not take care of relationship between elements in data structures nicely. For example, it seems that it cannot prove statically that the tail of a sorted list is also sorted unless programmers give axioms about sorted lists.

In a different line of work, Miranda [26], a statically typed functional programming language, provides datatypes with laws, which are rules to reconstruct data structures according to certain specifications. For example, we suppose that a datatype integer has three constructors Zero, Succ integer and Pred integer, and then a law converts Succ (Pred x) to x. More interestingly, Miranda can control application of laws by giving them conditional expressions. Using laws with conditionals, we can define lists which are sorted automatically. Both Miranda and our calculus provide a mechanism to convert data structures, but the purposes are different: in our work, type conversion is used only to check contracts, and so does not change "structures".

Systematic derivation of datatype definitions. As mentioned in Section 1, there is closely related work, in which systematic derivation of (indexed) datatype definitions is studied.

McBride [20] propose the notion of *ornaments*, which provide a mechanism to extend and to refine datatypes in a dependently-typed programming language. For example, the definition of lists can be derived from that of natural numbers by adding an element type; and the definition of lists indexed by their lengths can be derived. As far as we understand, he does not consider deriving new type definitions by changing the number of data constructors, as is the case for our work. Also, it is not clear whether partial refinements (the case where an index cannot be assigned to some values of the original datatype) can be dealt with in this framework. Partial refinements are important in our setting because our refinement types are for excluding some values in the underlying types.

Atkey et al. [3] developed derivation of inductive types from refinement types from a category-theoretic point of view. Moreover, it can deal with partial refinements. Our translation seems to be a concrete, syntactic instance of this framework. However, being abstract, their technique is not concerned about concrete representations of datatypes, which are significant when efficiency of casts is taken into account.

A similar idea is found in Kawaguchi et al. [17], who develop a refinement type system for static verification of programs dealing with datatypes. They allow programmers to write special terminating functions called measures, which will be used as hints to the verifier by indexing a datatype with the measure information.

Dependent and/or refinement type systems. The term "refinement types" seems to have many related but subtly different meanings in the literature. We use this term for types to denote subsets in some way or another. Refinement types are intensively studied in the context of static program verification.

In Freeman and Pfenning [12], datatypes can be refined by giving data constructors appropriate types. For example, one may give [] a special type null and cons a special type int → null → singleton_list, which means that, if cons takes an element and the empty list, then it yields a singleton list. Here, null and singleton_list are atomic type names. They did not allow refinement types to take arbitrary contracts to make type checking and type inference decidable. On the other hand, they combined refinement types with intersection types to express overloaded functionality of a single constructor.

Xi and Pfenning [30, 31] have designed and developed practical programming languages which support a restricted form of dependent types. Kawaguchi et al. [17] and Vazou et al. [27] have devised type inference algorithms for statically typed lambda calculi with refinement types and recursive refinements, which provides recursive types with refinements, and have implemented it for OCaml and Haskell, respectively. The refinements used there are derived from decidable languages such as (extensions of) Presburger arithmetic because their main focus is static verification. Our type system allows arbitrary Boolean predicates.

Our datatypes resemble *inductive datatypes* found in interactive proof assistants such as Coq [2] or Agda [1]. Aside from compatibility relation and casts, our syntax treats a formal argument $\langle x \rangle$ to a datatype to be parametric (notice that only argument types of data constructors have to be given). However, since x can appear in a refinement, conditions on the value of x can be enforced and so we do not lose much expressiveness.

6. Conclusion

We have proposed datatypes for manifest contracts with the mechanism of casts between different but compatible datatypes, and prove type soundness of a manifest contract calculus $\lambda^H_{\mathtt{dt}}$ with datatypes. In particular, the property that the value of a term of a refinement type satisfies the contract in the refinement type is proved for the first time in a purely syntactic manner. We have also given a formal translation from a refinement on lists to a datatype definition with refined constructors and proved the translation is correct. Moreover, the translation preserves the efficiency of casts.

As a proof of concept, we have implemented our casts using Camlp4. Our implementation does not support derivation of datatypes yet and a constructor choice function works by trial and error with backtracking but we are planning to extend the implementation with derivation of datatypes and an accompanying constructor choice function.

There are many directions of future work. First, we would like to investigate static contract checking using datatypes. A key theoretical property is upcast elimination: a property that removing upcasts—casts from a type to its supertype—does not change the behavior of a program in a certain sense, similarly to previous work [4, 18]. We expect refining constructor argument types is useful also for static checking [17]. Second, a proof that a generalized version of the translation given in Section 4 is correct remains as future work (although we do have translation). Third, it is worth investigating intersection types (or even Boolean operations) in this setting so that properties on data structures can be easily combined.

Acknowledgments

We thank Kohei Suenaga for valuable comments on an earlier draft. We are grateful to the anonymous reviewers for their fruitful comments and to Robby Findler for being our angel. Michael Greenberg encouraged us during the visit to our laboratory. This work was supported in part by Grant-in-Aid for Scientific Research (B) No. 25280024 from MEXT of Japan.

References

[1] The Agda 2 homepage. `http://wiki.portal.chalmers.se/agda/pmwiki.php`.

[2] The Coq proof assistant. `http://coq.inria.fr/`.

[3] R. Atkey, P. Johann, and N. Ghani. Refining inductive types. *Logical Methods in Computer Science*, 8(2:9):1–30, 2012.

[4] J. F. Belo, M. Greenberg, A. Igarashi, and B. C. Pierce. Polymorphic contracts. In *Proc. of ESOP*, volume 6602 of *LNCS*, pages 18–37, 2011.

[5] G. M. Bierman, A. D. Gordon, C. Hriţcu, and D. Langworthy. Semantic subtyping with an SMT solver. In *Proc. of ACM ICFP*, pages 105–116, 2010.

[6] M. Blume and D. A. McAllester. Sound and complete models of contracts. *J. Funct. Program.*, 16(4–5):375–414, July 2006.

[7] O. Chitil. A semantics for lazy assertions. In *Proc. of ACM PEPM*, pages 141–150, 2011.

[8] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, Sept. 1992.

[9] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proc. of ACM ICFP*, pages 48–59, 2002.

[10] R. B. Findler, S. Guo, and A. Rogers. Lazy contract checking for immutable data structures. In *Proc. of IFL*, volume 5083 of *LNCS*, pages 111–128, 2008.

[11] C. Flanagan. Hybrid type checking. In *Proc. of ACM POPL*, pages 245–256, 2006.

[12] T. Freeman and F. Pfenning. Refinement types for ML. In *Proc. of ACM PLDI*, pages 268–277, 1991.

[13] M. Greenberg. *Manifest Contracts*. PhD thesis, University of Pennsylvania, 2013.

[14] M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In *Proc. of ACM POPL*, pages 353–364, 2010.

[15] J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, pages 93–104, 2006.

[16] D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. In *Trends in Functional Prog. (TFP)*, 2007.

[17] M. Kawaguchi, P. M. Rondon, and R. Jhala. Type-based data structure verification. In *Proc. of ACM PLDI*, pages 304–315, 2009.

[18] K. Knowles and C. Flanagan. Hybrid type checking. *ACM TOPLAS*, 32(2:6):1–34, Feb. 2010.

[19] K. Knowles, A. Tomb, J. Gronski, S. N. Freund, and C. Flanagan. Sage: Unified hybrid checking for first-class types, general refinement types, and dynamic (extended report). Technical report, UCSC, 2007.

[20] C. McBride. Ornamental algebras, algebraic ornaments. *J. Funct. Program.*, 2014. To appear.

[21] B. Meyer. *Object-Oriented Software Construction, 1st Edition*. Prentice-Hall, 1988. ISBN 0-13-629031-0.

[22] P. C. Nguyen, S. Tobin-Hochstadt, and D. Van Horn. Soft contract verification. In *ACM ICFP*, 2014.

[23] B. C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.

[24] P. M. Rondon, M. Kawaguchi, , and R. Jhala. Liquid types. In *ACM PLDI*, 2008.

[25] J. G. Siek and P. Wadler. Threesomes, with and without blame. In *Proc. of ACM POPL*, pages 365–376, 2010.

[26] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proc. of ACM FPCA*, volume 201 of *LNCS*, pages 1–16, 1985.

[27] N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In *Proc. of ESOP*, volume 7792 of *LNCS*, pages 209–228, 2013.

[28] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton Jones. Refinement types for Haskell. In *ACM ICFP*, 2014.

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

[30] H. Xi. Dependent ML: An approach to practical programming with dependent types. *J. Funct. Program.*, 17(2):215–286, Mar. 2007.

[31] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proc. of ACM POPL*, pages 214–227, 1999.

[32] D. N. Xu. Hybrid contract checking via symbolic simplification. In *Proc. of ACM PEPM*, pages 107–116, 2012.

[33] D. N. Xu, S. L. Peyton Jones, and K. Claessen. Static contract checking for Haskell. In *Proc. of ACM POPL*, pages 41–52, 2009.