Abstract Machines for Dynamic Computation

Christopher David Walton

Doctor of Philosophy
University of Edinburgh
2001

Abstract

In this thesis we address the challenges associated with the provision of *dynamic* software architectures. These are systems in which programs are constructed from separately compiled units with a facility for the replacement of these units at runtime. Typical examples of applications which will benefit from this dynamic approach are long-lived systems in which downtime is highly undesirable, for example, web-servers, database engines, and equipment controllers. In addition, dynamic software architectures are also gaining popularity with the recent advent of wide-area Internet applications, where it is often impractical to compile a program in its entirety or begin execution in a single step.

Our approach to dynamic software architectures differs from earlier attempts in that we guarantee the safety of the replacement operation. This is done by founding our techniques on the rigour of strong typing. In the first half of the thesis we take an existing static software architecture with strong typing facilities and modular program construction, namely the Standard ML platform, and equip it with facilities for separate-compilation and code-replacement of modules. The resulting dynamic software architecture, which we call Dynamic ML, ensures the safety of replacement through an effective use of state-of-the-art advances in the fields of types in compilation and abstract machines.

In the latter half of the thesis we extend Dynamic ML with a facility for distributed execution and adapt our code-replacement model accordingly. This will permit the construction of larger dynamic architectures, for example, across a distributed network of workstations. We also perform a mechanical verification of the distributed algorithm by model checking, to gain further confidence in the correctness of our approach. At the end of the thesis we outline an implementation of our techniques for the Java language, demonstrating the portability of our approach.

Acknowledgements

First and foremost I would like to thank my supervisor Stephen Gilmore. He was generous with his time and provided invaluable help and support throughout. I was very fortunate to work in the LFCS at Edinburgh, a place full of friendly, talented, and helpful people. In particular, special thanks are due to Mike Fourman, and my colleagues Dilsun Kırlı and Kostas Tourlas. I am also grateful to the EPSRC for providing the funding for my work. This thesis is dedicated to my parents, who have always been a supportive influence in my life.

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, with the exception that some parts of Chapter 7 were composed in collaboration with Stephen Gilmore and Dilsun Kırlı, and where explicitly stated otherwise in the text.

Parts of this thesis have already been published elsewhere. Earlier versions of Chapter 6 have appeared in [Wal99] and [Wal00], and early revisions of Chapter 7 appeared in [WKG98] and [WKG00].

(*Christopher D. Walton*)

Table of Contents

Chapter 1

Introduction and Synopsis

Traditional software architectures are *static* in nature. Programs are treated as atomic entities; designed to be compiled as a whole and executed to completion. In this thesis, we explore the relatively new field of *dynamic* software architectures, in which programs are constructed from separately compiled units and updated during runtime.

One particular application which would benefit from a dynamic approach is a web server. The global nature of the World Wide Web demands that many of these servers remain operational for 24 hours, 7 days a week. This presents an interesting challenge to the web server administrator who wishes to seamlessly upgrade the web server, or migrate the server to a new machine. Similar problems exist in many long-lived systems where software maintenance is required without interruption of a service. While 'hot' upgrading of hardware architectures has become commonplace, corresponding techniques for software architectures have not been fully developed. Typically, arcane ad-hoc solutions are employed which provide few, if any, guarantees of safety, e.g. [SF93].

The benefits of dynamic architectures are not limited to long-lived systems. A broadening class of systems and applications are beginning to exhibit similar properties in an effort to provide customisation and extensibility. In particular, with the advent of wide-area 'Internet' applications, where it is rapidly becoming impractical to compile a program in its entirety or begin execution in a single step. Such applications require enhanced flexibility which traditional static architectures are struggling to provide.

In an effort to add some degree of software dynamism to static architectures, runtime extensions have recently become available in several operating systems: shared libraries in UNIX, and dynamically linked libraries (DLLs) in Windows. In a similar vein, the component object models CORBA [Obj98] and DCOM [Red97] contain runtime component loading facilities. However, these extensions do not provide the full flexibility offered by dynamic architectures, e.g. general code replacement. Moreover, these extensions are rather flimsy; missing libraries and incompatibilities between versions can readily lead to system chaos.

A better approach is offered by the Java platform [Kra96]. Java offers dynamic loading and execution of compiled classes through the `ClassLoader` mechanism. Java also allows a restricted form of code replacement called *binary-compatibility*. This allows the definition of a class to be replaced with another at runtime, though it does not update any objects which are based on the old definition of the class. The Java Language Specification [GJS96] defines a set of properties, which must be satisfied in order for a class (or interface) to be considered a valid binary-compatible replacement. However, it has been demonstrated that these properties are flawed [DEW99]; it is possible to perform a sequence of program modifications, each of which are binary compatible in the Java sense, and produce a program which cannot be executed.

The Erlang programming language [AVWW96] offers true dynamism in that arbitrary program modules may be loaded and replaced at runtime. Erlang has enjoyed moderate success in programming long-lived systems in the telecommunications industry. However, code replacement in Erlang is not altogether robust. Code replacement is unrestricted in that any module may be replaced with any other. This produces a very general model of code replacement, but provides no protection from runtime failures.

A different kind of dynamism can be provided through an extension of persistent programming language features. In such languages, program code and data is stored in a persistent database, enabling programs to be readily halted and restarted as required. Code replacement can be provided in such languages by performing meta-level transformations on the persistent database between program runs. This technique is flexible in that persistence can often be included in existing programming languages with only minimal changes. In [KMM97] a form of code-replacement, called *application evolution*, is defined and an implementation sketched for the persistent object-serialisation facility of Java.

In contrast with the above systems, we take an approach to software dynamism founded on strong typing, making full use of modern developments in the areas of types in compilation and abstract machines. The main focus of attention in this thesis is the Standard ML programming language [MTHM97], though an implementation for the Java programming language is discussed in the concluding chapter. We will define an extension to Standard ML, hereafter called Dynamic ML, with a facility for run-time code-replacement.

1.1 Modular Code Replacement

To facilitate the construction of large programs, Standard ML has a sophisticated modules system. Standard ML modules, called *structures*, provide flexible encapsulation through interfaces called *signatures*, and flexible assembly by parametric modules called *functors*. In order to facilitate the replacement process, and to indicate a suitable size of component for replacements, we will fix on *first-order module-level* replacement. That is, we allow the replacement of signatures by other signatures and structures by other structures, under conditions which we will discuss later. We treat functors as purely compile-time objects, used only to generate structures, and so we do not present a replacement model for them. Semantically, module-level replacement is an apt choice because it facilitates the improvement of programs by revision and replacement of data structures in tandem with the creation and access functions which are associated with them. In implementation terms, module-level replacement is supported by the facility to compile Standard ML modules in isolation. Thus an executing program may be upgraded without the need to re-compile it in its entirety.

A runtime code-replacement using the modules system of ML was first proposed in [App94]. This paper presents a rough outline of a technique for replacing one module with another at runtime. The technique hinges on a flexible runtime linking operation. The replacement of one module with another is performed by defining a cut-off point where the usage of the old module is discontinued and usage of the new module begins. Though, as there may still be instances of functions and data values which depend on the old definition, the old module code is retained for as long as necessary. However, this proposal has remained a rough outline and has never been formalised or implemented. Although the underlying concepts are sound, there is a problem with the implementation in that permitting the use of transparent signatures means that code may depend on internal representation types used within modules, and will break the program when the new module definition is introduced.

We define a more advanced form of code-replacement in ML where introducing a new module definition results in the complete update of any functions and data values which were constructed with the old definition. After a replacement, the old definition is entirely removed from the system. The main advantage of our method is that a replacement which fixes an error in the program or enhances the performance will have an immediate effect. Furthermore, this approach is more space-efficient in that it is not necessary to keep copies of old module definitions around indefinitely. However, it is worth noting that we must take steps to prevent the replacement of any function which is currently being executed. If allowed, such unguarded replacements would result in undefined behaviour as the continuation point for the computation would no longer exist. Consequently, our technique only permits the replacement of functions that are inactive. We will now illustrate our replacement technique by means of an example.

As a running example we consider the replacement of one implementation of a name table with another which is functionally equivalent but offers improved performance. For simplicity we choose the type of names to be character strings. The table type is abstract and in moving from the inefficient implementation to the more efficient one we are in fact replacing an unsorted list with a binary search tree. Both implementations match the `TABLE` signature shown in Figure 1.1. That is, both implementations provide an abstract type for tables, a declaration of a type of names to be implemented by strings, a constant value denoting the empty table and functions to insert names and the test for membership in the table. Standard ML structures provide a dot notation for accessing the components of structures and thus structures named `Table` matching the `TABLE` signature would define types `Table.name` and `Table.table`, a constant `Table.empty` and functions `Table.insert` and `Table.member`.

```
signature TABLE =
  sig
    type name = string
    type table
    val empty : table
    val insert : name * table -> table
    val member : name * table -> bool
  end;
```

Figure 1.1: *Signature for the* `Table` *Module.*

The implementations are shown in Figure 1.2 and Figure 1.3. Matching against the signature, using the `:>` syntax, is *opaque* and thus outside the structure body we cannot make use of the typing information which is known inside the structure body. For example, we cannot make use of the fact that in the first version of the structure tables are implemented as Standard ML lists. This prevents us from applying functions defined on lists, such as `List.rev`, to values of type `Table.table`. Inside the structure body we are free to use list constants such as the empty list, denoted by two square brackets, and list constructors such as cons, denoted by two colons.

```
structure Table :> TABLE =
  struct
    type name = string
    type table = name list
    val empty = []
    fun insert (s, t) = s :: t
    fun member (s, []) = false
      | member (s, h :: t) = s = h orelse member (s, t)
  end;
```

Figure 1.2: *Inefficient Implementation of Tables.*

After a careful comparison of the two implementations, we should be able to agree that the second implementation could be used as a replacement for the first. A specification-based analysis of the two implementations would judge them to be behaviourally equivalent. Less prosaically, we could think that we would not compute any different results if we had initially built our system using the second implementation of the structure instead of the first. Our notion of structure replacement certainly includes logically undetectable replacements such as this one but it is more lax, additionally allowing replacements which change the observable behaviour of the program under modification. Examples of the latter would include replacing a structure with a version which logged function

```
structure Table :> TABLE =
  struct
    type name = string
    datatype table = empty
           | node of table * name * table

    fun insert (s, empty) = node (empty, s, empty)
      | insert (s, node (l, v, r)) =
          if s < v then node (insert (s, l), v, r)
          else if s > v then node (l, v, insert (s, r))
               else node (l, v, r)

    fun member (s, empty) = false
      | member (s, node (l, v, r)) =
          if s < v then member (s, l)
          else if s > v then member (s, r)
               else true
  end;
```

Figure 1.3: *An Improved Implementation.*

calls, perhaps in order to extract statistical information about the program's runtime performance or in order to aid with the detection of logical errors in the implementation.

The formal requirement which structure replacements must satisfy can be captured by static type-checking: a replacement `S2` for a structure `S` must match every signature constraint which `S` matched in the original program. In practice, this means that the replacement structure must not omit any functions, types or values which were exported by the structure which it replaces. Functions, values and types which were defined and only used internally may be omitted in the implementation of the replacement structure. For simplicity here, we do not provide a facility for combining replacement with renaming. Structures and signatures replace structures and signatures which have the same name.

Signature replacement is a facilitating operation which allows more permissive signatures to replace more restrictive ones. An effect of this can be to make visible functions, types and values which had been hidden by the application of a signature constraint. Replacement such as this is subject to type preservation conditions which constrain the relationship between the signatures. Signature replacement must not cause changes of typing information about visible values and functions. It might be more appropriate to term this operation *signature extension* since it will be most often used to allow the declarations in a structure to be supplemented by others which increase the functionality of any matching structure.

Consider the situation where we replace our `TABLE` signature with one which includes the function specification shown in Figure 1.4. That is, upon being applied to a pair of a table and an output stream such as `TextIO.stdOut` the function will serialise the contents of the table to the output stream and return a unit value, of type `unit`, to signal completion. We could not replace our previous version of the `TABLE` signature with one extended by the check-pointing function without first upgrading all structures which match this signature to contain an implementation of the function. This would not be visible under the old signature but would become visible when we upgrade the signature to include it. We could then replace client structures of the `TABLE` structure to allow them to make calls to the newly added check-pointing function.

```
val checkpoint : table * TextIO.outstream -> unit
```

Figure 1.4: *A Check-pointing Function.*

1.1.1 Expressing Replacement

Our method of code replacement is intended to be suitable for updating programs where the application programmer has followed good software engineering practice by encapsulating information such as the concrete representation of data structures. This disciplined approach to programming facilitates our replacement of a list by a tree, ensuring that the change is invisible to the users of the `Table` structure. However, this proves to be a disadvantage when we come to consider the problem of describing the replacement of values of the old datatype with values of the new datatype. Specifically, the constructors of the old concrete representation are not visible, due to the encapsulation which is provided by the application of the `TABLE` signature constraint to the `Table` structure in our example.

In order to circumvent this difficulty we could abstract over a `Table` structure which is specialised to implement a name table as a list of character strings. Structure abstractions such as these are Standard ML functors. Given a structure matching the specialised `TABLE` signature the functor body could describe a structure which implements name tables as binary search trees. In addition the structure could contain functions to convert from the types of the given structure to the types of the new. We place the conversion functions inside an `Install` structure and we follow a convention of mapping values from their old representation to their new one using functions which have the same identifier as the type which they update. Such duplicate use of identifiers is possible in Standard ML because the language maintains different name spaces for different categories of identifiers (value constructors, type constructors and record labels in the core language and signature, structure and functor identifiers in the modules language).

Figure 1.5 shows this method of structure replacement encoded as a Standard ML functor. The functions to update values of type `name` and `table` to use the new types are respectively the identity function, mapping `x` to `x`, and an application of the Standard ML Basis library function implementing folding a function across a list with right associativity. The `List.foldr` function is applied to the `insert` function for trees and the `empty` tree, specialising it to provide a function which maps lists to binary search trees.

The expressive power which Standard ML functors provide is sufficient to allow us to state our wish to replace lists by binary search trees but it would not be sufficient to allow us to subsequently replace these trees with, say, balanced trees. The reason is this: in Standard ML the type expression which appears in a qualification of a signature expression may only refer to type constructors which are in the scope of the signature expression. The nullary type constructor `string`

```
functor InstallTable (structure Table : TABLE where
                      type table = string list ) :> TABLE =
  struct
    type name = string
    datatype table = empty
                   | node of table * name * table

    fun insert (s, empty) = node (empty, s, empty)
      | insert (s, node (l, v, r)) =
          if s < v then node (insert (s, l), v, r)
          else if s > v then node (l, v, insert (s, r))
               else node (l, v, r)

    fun member (s, empty) = false
      | member (s, node (l, v, r)) =
          if s < v then member (s, l)
          else if s > v then member (s, r)
               else true

    structure Install =
    struct
      val name : Table.name -> name = fn x => x
      val table : Table.table -> table = List.foldr insert empty
    end
  end;
```

Figure 1.5: *Functor For Replacing Tables.*

and the unary type constructor `list` are in scope by virtue of being pre-defined in the language. However, the binary search tree datatype which we defined in the body of the `InstallTable` functor is not pre-defined. Further, it is not exported from the structure which is formed by applying the functor to the old version of the `Table` structure due to the opaque signature matching. For this reason, we include in Dynamic ML an extended version of Standard ML's `where type` qualification, illustrated in Figure 1.6. The `where datatype` qualification enables us to express the replacement of local datatypes with alternate definitions.

```
functor InstallTable (structure Table : TABLE
                        where datatype table = empty
                        | node of table * name * table) :> TABLE
```

Figure 1.6: *The* `where datatype` *Qualification.*

1.2 Types and Type Systems

A *type* is defined as a collection of values that share some computationally-checkable property, e.g. a type of integer values. Program variables are assigned values during execution. A programming language which allows variables to be restricted to certain types is called a *typed language*, while a language which does not restrict the range of variables is called an *untyped language*. We will only be concerned with typed languages in this thesis as they provide a far greater degree of predictability of behaviour than untyped languages, e.g. LISP, Scheme, PHP.

In most typed languages, a process called *type checking* is performed. A program that passes the type checking is said to be *well typed*. Type checking can be done at compile-type, called *static typing*, e.g. Standard ML, or at runtime, called *dynamic typing*, e.g. Erlang. The fundamental purpose of type checking is to prevent the occurrence of execution errors. In static typing, we perform a compile-time check that each program fragment, i.e. statement, expression, etc. defines an element of a type that is specified in the program text, or inferred from the context. A program which fails to type check in this manner is rejected by the compiler and may not be executed. Dynamic typing similarly tries to prevent the execution of erroneous operations, but the check is performed while the program is executed. Dynamic typing is more forgiving in that only errors which actually occur are detected. To illustrate the difference between static and dynamic typing, we consider the following program fragment:

```
if X then 1 else 2 + "hello"
```

The addition `2 + "hello"` would be considered an error if 2 has integer type and `"hello"` has string type. This error would only be detected by dynamic typing if the conditional expression `X` evaluated to `false`. At compile-time, when static-typing is performed, it is usually impossible to determine the value of the arbitrary expression `X`. Thus, both branches of the conditional expression are checked and the error would always be detected. Static typing always detects errors earlier than dynamic typing, and may also detect errors that would otherwise remain hidden.

The static and dynamic typing disciplines are not altogether mutually exclusive. There have been a number of hybrid schemes which combine dynamic types with static typing. The most popular of these languages is Java, which performs static type checking augmented with runtime checks, e.g. on array bounds and coercions. The Amber language [Car86] was the first ML-like language to include dynamic types with static type checking. The primary motivation for including dynamic types is that it allows for structured I/O, and enables the definition of straightforward interfaces to other programming languages. The technique hinges on the addition of a special type, called a *dynamic*, to the language. A dynamic is a pair of a value v and a type expression τ, such that v has type τ. There are essentially just two operations required in a language for operating on dynamics: a constructor `dynamic` for building objects of type `Dynamic`, and a `typecase` operation for operating on the type of a dynamic. The technique for including dynamics in the first-order types of ML was presented in [ACPP91]. This was later extended to include polymorphic types in [LM93] and [ACPR95]. These extensions permit ML programs such as the example in Figure 1.7 to be constructed. In this example, the type of the `read` operation is determined at runtime, rather than compile-time.

```
fun displayFile (filename : string) =
      let val image = read (filename)
      in
        typecase image of
          (v : Vector) => displayVector v
          (b : Bitmap) => displayBitmap b
          (s : string) => displayString s
          _  => raise UnknownType
      end;
```

Figure 1.7: *Dynamic Typing in ML Example.*

The inclusion of dynamic types in an ML-like language allows a number of language extensions to be readily defined. For example, a restricted form of code-replacement, called meta-programming, is defined in [Dor98], and a form of dynamic code-mobility is described in [Dug96]. Nonetheless, we have decided against using dynamic types in this thesis. Although a solution founded on dynamic types would be perfectly valid, we adopt an approach that we believe to be superior based on static typing and dynamic linking. We feel that dynamic types simply shift the burden of code-replacement from the language designer to the programmer. Furthermore, the introduction of dynamic types removes the predictability of behaviour, and hence safety, from the model. Finally, dynamic typing introduces significant overheads on evaluation, due to the requirement for runtime consistency checks.

Our model of code-replacement is based on static types and all checks for correctness are performed at compile time. Replacement is accomplished by a dynamic linking operation defined in Chapter 7. Furthermore, only minor extensions to the definition of the ML module system are required. In principle, existing Standard ML programs can be used for code-replacement without requiring any source-level changes.

A *type system* formalises the type rules of a programming language. In general, the type system of a programming language is specified separately from the algorithm used to perform type-checking. It is easier to explain the typing aspects of the language by a type system rather than an algorithm. Moreover, there may be many type-checking algorithms for a single type system [Ler92]. In this thesis, we will only deal with type systems rather than type-checking algorithms. Our type systems are formulated in the style of [Car97]. Two descriptions of type-checking algorithms are presented in [Car87] and [Pey87].

A program which does not generate any runtime type errors is called *type safe*. A static type system should reject, at compile-time, all programs which are not type safe. A static type system with this property is said to be *sound*, and a programming language with a sound static type system is said to be *strongly typed*. It is worth noting that the static prediction of all runtime type errors is an undecidable property of programs. Thus, any sound type system will also reject some programs that would not generate runtime type errors. However, it is generally possible to minimise these rejections and therefore generate a useful type system. In reality, certain statically checked languages do not ensure type safety. These languages are called *weakly typed* meaning that some unsafe operations are detected while others are not. For example, the C programming language has many unsafe features such as casting and pointer arithmetic. Many of these deficiencies have been addressed in Java, but some have not [DEK99].

Languages in the Standard ML family are strongly typed. In order to enforce the application of the type-checking stage, these languages make a strict distinction between *elaboration* and *evaluation*, insisting that programs which have not successfully elaborated cannot be evaluated at all. The rigid ordering of these two stages prohibits the execution of any programs which attempt to use data values in ways which are not allowed by their type and thus eliminates a large number of software errors which would manifest themselves at runtime if working in an untyped programming language. However, several authors have observed that two stages are not enough for complex applications such as program generators. This has led to approaches such as the *multi-stage* programming paradigm for MetaML [TS97], and the *staged type inference* paradigm [SSP98]. Dynamic ML retains the distinction between elaboration and evaluation but enhances the flexibility of the model by allowing safe runtime updates.

1.2.1 Type-Preserving Compilation

The compilation of functional programming languages is typically performed by a sequence of translations between intermediate representations. Figure 1.8 illustrates the compilation phases and intermediate forms of a typical ML compiler. The compiler is divided into the *front end* which deals directly with the source language, and the *back end* which is responsible for optimisation and code-generation. The compiler back-end generally attracts the majority of compiler research as it has the greatest impact on the performance of the final code. The representation upon which optimisations are performed is typically an enriched variant of the λ-calculus.

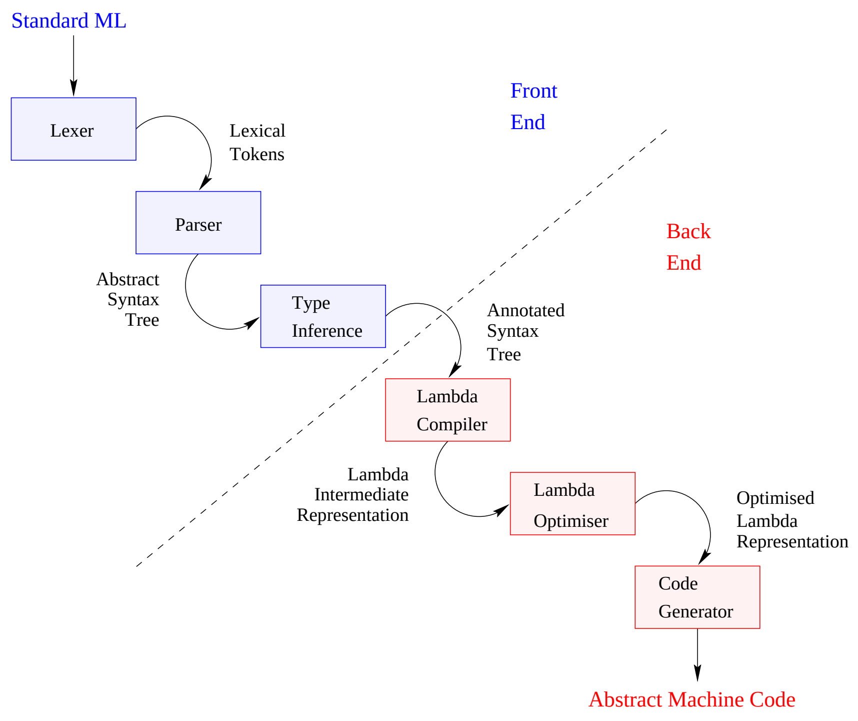


Figure 1.8: *Standard ML Compiler Overview.*

Until recently, type information was only present in the front-end of the compiler: after the type-inference phase the type information was considered an unnecessary overhead and was discarded. This also allowed the reuse of common untyped LISP-style compiler back-ends. However, modern compiler research has highlighted the fact that this type information can also be useful in the compiler back-end [Ler98]. Compilers that propagate type information into the back-end are called *type-preserving* compilers. Most of the current Standard ML compilers are type preserving, e.g. Standard ML of New Jersey [Sha97], the ML-Kit [TBE$^+$98], TILT [TMC$^+$96], and MLj [BKR98].

In type-preserving compilers, the intermediate representations are predominantly based on variants of the typed λ-calculus. Type-checking may be performed at each stage of the compilation process. This is used to ensure that the translations between intermediate representations and the code optimisations have preserved the safety of the code. The structure of a type-preserving compiler back-end is illustrated in Figure 1.9.

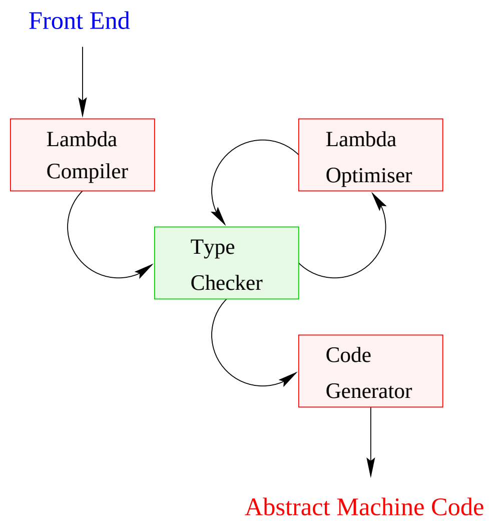


Figure 1.9: *Type-Preserving Compiler Back-End.*

Typed intermediate representations are particularly useful for compiler development as the type-checking acts as a debugging tool [BKR98]. However, the type information is also useful for guiding a number of optimisations. For example, the compiler can choose to represent certain types of data more efficiently, e.g. arrays, rather than using a universal ‘boxed’ representation. An aggressively optimising compiler may use several different typed intermediate representations and perform a range of optimisations on each.

Typed intermediate representations are also useful for describing transformations on the Standard ML language. By basing a transformation on a suitable intermediate representation, it is possible to obtain an operation that is applicable to the whole of Standard ML while avoiding a great deal of unnecessary complexity. For example, pattern matching may appear in the intermediate language as simple switch statements, having been converted by a higher-level match compiler. We use this approach to define code replacement and translation in this thesis. In Chapter 2 we define an intermediate language, called $\mathcal{M}\Lambda$, which is powerful enough to represent the whole of Dynamic ML. We define a sound type-system for the language in Chapter 3 which we use as the foundation for code-replacement.

1.3 Abstract Machines

A type system defines the *static semantics* of a programming language. It is important to note that a type system is not an arbitrary collection of rules. Well-typing is intended to correspond to a semantic notion of good program behaviour. The execution behaviour of a programming language is defined by the *dynamic semantics* of the language. Type-soundness essentially states that the static and dynamic semantics of the language are consistent.

In the formal presentation of dynamic semantics there is a much greater variance in style than within static semantics, primarily dependent on the properties of the language being described. However, the two foremost semantic styles are *operational* and *denotational*. Operational semantics is concerned with defining *how* programs are executed, in particular, specifying the exact sequence of evaluation steps associated with the execution of a program. In contrast, denotational semantics is concerned with *what* mathematical function a program computes, without saying explicitly how it is to be computed. The two methods are complementary. Roughly speaking, the operational semantics are most useful for specification and implementation, while the denotational semantics are most useful for proofs of properties of a language. Since the emphasis of this thesis is on providing a specification for a dynamic software architecture, we shall be concerned only with the operational technique. Expressing parallelism and non-determinism, as required by a dynamic architecture, is considered harder in a denotational semantics. Note that it is perfectly possible to prove properties of a language using the operational technique. However, the resulting proofs are generally more cumbersome than their denotational equivalents.

There are two different styles of operational semantics, distinguished by the kind of mathematical relation which they use. In a *relational* semantics, we define a relation between the initial and final states of a whole program fragment. By contrast, in a *structured-transition* semantics, the relation is defined only between the current and next computational step. In order to illustrate the difference between these two approaches, we define sequential composition in a relational style (Figure 1.10), and in a structured-transition style (Figure 1.11).

$$\frac{\sigma \vdash p_1 \Downarrow \sigma' \qquad \sigma' \vdash p_2 \Downarrow \sigma''}{\sigma \vdash p_1 ; p_2 \Downarrow \sigma''} \tag{1.1}$$

Figure 1.10: *Relational Semantics*

The relational rules have the general form $\sigma \vdash p \Downarrow \sigma'$ which should be read: in state σ, the evaluation of the program fragment p yields a new state σ'. Typically, the state σ is used to hold the values of the bound variables. The structured-transition rules have the form $\langle p,\ \sigma\rangle \rightarrow_1 \langle p',\ \sigma'\rangle$ which should be read: one step of the evaluation of p in state σ yields a new program fragment p' and a new state σ'.

In the relational semantics, the sequential evaluation of both program fragments p_1 and p_2 is captured by a single rule. By contrast, in the structured transition semantics, the evaluation will result in a long chain of steps for each program fragment (Rule 1.2), terminating with ϵ (Rule 1.3).

$$\frac{\langle p_1,\ \sigma\rangle \rightarrow_1 \langle p_1',\ \sigma'\rangle}{\langle p_1; p_2,\ \sigma\rangle \rightarrow_1 \langle p_1'; p_2,\ \sigma'\rangle} \tag{1.2}$$

$$\frac{\langle p_1,\ \sigma\rangle \rightarrow_1 \langle \epsilon,\ \sigma'\rangle}{\langle p_1; p_2,\ \sigma\rangle \rightarrow_1 \langle p_2,\ \sigma'\rangle} \tag{1.3}$$

Figure 1.11: *Structured-transition Semantics*

In this thesis, we will only be concerned with the relational approach. The primary reason for this choice is that the structured-transition approach requires a large number of rules for even the simplest of operations. This would become extremely cumbersome for a language such as Dynamic ML. In general, the advantage of the structured-transition approach is that it allows concurrency to be easily expressed: each rule is treated as an atomic action in the computation, concurrency is then introduced simply by interleaving these atomic actions. Since we will require concurrency in Dynamic ML, it may appear that we have no choice. However, in Chapter 8, we demonstrate that concurrency can be expressed in a relational semantics using *events*.

A form of relational semantics, called *natural semantics* [Kah87], is used in the Definition of Standard ML [MTHM97], so called because the evaluation of the rules is reminiscent of Natural Deduction. Unfortunately, this style of semantics is too high-level for defining Dynamic ML. The state (σ in our examples above) hides all the details of memory management, for example, how data objects should be represented in memory, and the memory location at which they reside. However, for dynamic code replacement, these are precisely the details which we wish to expose. We therefore adopt a more discriminating relational style called an *abstract-machine* semantics.

In the abstract machine style we define relations, called *transitions*, between machine states (also called *configurations*). The machine state is generally represented by a tuple of stacks or finite-maps. For example, the state of the SECD abstract machine [Lan64] is a four-tuple consisting of a Stack, an Environment, a Control stack, and a Dump stack, hence the name SECD. The Stack is used to hold intermediate results when computing the value of expressions, the Environment is used to hold variables bound during execution, the Control stack is used to hold the program being executed, and the Dump stack is used to save values during function calls. Execution of the SECD machine is defined by transitions between machine states: $(S,\ E,\ C,\ D) \Rightarrow (S',\ E',\ C',\ D')$. The sequential composition of two program fragments is therefore defined as in Figure 1.12. We evaluate the control sequence $p_1 \cdot p_2 \cdot C$, where C is the remaining program and the dot symbol is the stack element separator.

$$\frac{\begin{array}{l}(S,\ E,\ p_1 \cdot p_2 \cdot C,\ D) \Rightarrow (S',\ E',\ p_2 \cdot C,\ D') \\ (S',\ E',\ p_2 \cdot C,\ D') \Rightarrow (S'',\ E'',\ C,\ D'')\end{array}}{(S,\ E,\ p_1 \cdot p_2 \cdot C,\ D) \Rightarrow (S'',\ E'',\ C,\ D'')} \tag{1.4}$$

Figure 1.12: *SECD Abstract Machine Semantics*

The abstract machine style is clearly much closer to an actual implementation than the previous relational form. In fact, it is relatively straightforward to construct a program which performs identically to an abstract machine definition. For this reason, there has been some research into automating the translation from natural-semantics into the abstract machine style for the purpose of compiler generation, e.g. [Die96, Pet99].

There are many differing configurations of abstract machines, largely determined by the language which they represent. For example, the SECD abstract machine above is designed for executing simple LISP programs and was implemented in the LispKit compiler [Hen80]. Similarly, the Functional Abstract Machine (FAM) [Car83] is an abstract machine designed for the execution of strict functional programming languages, such as Standard ML. A direct implementation of the FAM appeared in the Edinburgh ML compiler. For lazy functional languages such as Haskell, an abstract machine based on graph-reduction is generally used, for example the G-machine [Pey87]. The Java Virtual Machine (JVM) [LY96] can also be considered an implementation of an abstract machine for an object-oriented language.

1.3.1 Type-Preserving Evaluation

In a type-preserving compilation, type information is present throughout the *compilation* process. This enables a range of optimisations that are not possible using untyped representations. Extending this idea further, we can propagate the type information into the *evaluation* process. Once again, we find that this provides a number of benefits which cannot be readily achieved with an untyped evaluation, namely *tag-free* garbage collection, and *code-replacement*. Further benefits such as type-based security schemes, run-time profiling, debugging and optimisation are also possible, though we do not discuss them here.

The propagation of type information into the evaluation process is a relatively new concept, previously dismissed as too expensive for use in a real implementation. However, recent developments in abstract machine technology, exemplified by the Java Virtual Machine [LY96] (which retains class and method typing information at run-time) have shown that efficient implementations are certainly possible. While we favour clarity of definition over efficiency of implementation in this thesis, we note that similar techniques, such as just-in-time compilation, can readily be applied to our abstract machine definitions.

We will define type-preserving evaluation by means of a typed abstract machine definition. In this abstract machine, each stage of the evaluation yields both a type and an associated value. As an example, we define two rules of a naive type-preserving variant of the SECD abstract machine in Figure 1.13. These rules may be used to perform the addition of two integers. In Rule 1.5, the evaluation of the instruction $INT\ i$ yields a pair containing the type `int` and the value i on the stack. In Rule 1.6, the addition of two expressions is performed. Note that both expressions evaluate to integer values. Thus, the sequence of instructions $INT\ 1 \cdot INT\ 2 \cdot IADD$ would yield the pair (`int`, 3) on S.

$$\frac{}{(S,\ E,\ INT\ i \cdot C,\ D) \Rightarrow ((\texttt{int},\ i) \cdot S,\ E,\ C,\ D)} \tag{1.5}$$

$$\frac{\begin{array}{l}(S,\ E,\ e_1 \cdot C,\ D) \Rightarrow ((\texttt{int},\ v_1) \cdot S,\ E,\ e_2 \cdot IADD \cdot C,\ D) \\ ((\texttt{int},\ v_1) \cdot S,\ E,\ e_2 \cdot IADD \cdot C,\ D) \Rightarrow \\ \quad ((\texttt{int},\ v_2) \cdot (\texttt{int},\ v_1) \cdot S,\ E,\ IADD \cdot C,\ D)\end{array}}{(S,\ E,\ e_1 \cdot e_2 \cdot IADD \cdot C,\ D) \Rightarrow ((\texttt{int},\ v_1 + v_2) \cdot S,\ E,\ C,\ D)} \tag{1.6}$$

Figure 1.13: *Typed SECD Abstract Machine*

In Chapter 5 we define a typed abstract machine semantics for our $\mathcal{M}\Lambda$ language. For our purposes, the type-preserving abstract machine approach provides exactly the right level of detail. The state of the abstract machine makes explicit the structures required for garbage collection and code replacement, as illustrated in Figure 1.14. The abstract machine definition itself can be used to show the correctness of the code replacement, and the abstract machine definition could easily serve as the basis for an implementation of Dynamic ML.

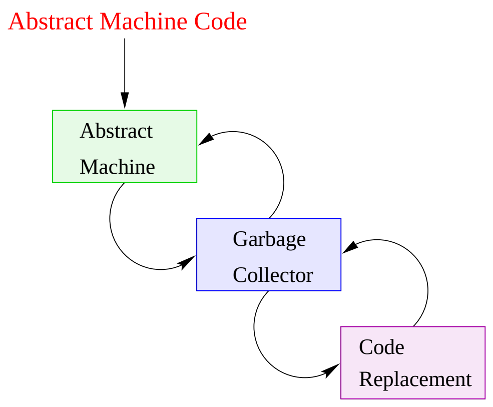


Figure 1.14: *Type Preserving Run-Time System.*

We note that type-preserving evaluation becomes significantly more complex in the presence of polymorphic types. It becomes necessary to provide explicit type environments for binding type variables to types. This clearly adds a layer of complexity to the abstract machine definition which would incur a significant runtime penalty in an implementation. A possible solution would be the elimination of all polymorphism from the language at compile-time. Each polymorphic function could be specialised, producing a separate version for each type instance at which it is used. A scheme for eliminating polymorphism in this manner for Standard ML is detailed in [Els98], and a similar scheme is employed by the MLj [BKR98] compiler. Unfortunately, this scheme is incompatible with code replacement, as we can clearly no longer determine at compile-time all of the type instances for a function. We therefore define a scheme which manipulates polymorphic types in a lazy manner, which we call *deferred instantiation*, in Chapter 5. This scheme may be used to provide an efficient implementation of our type-preserving abstract machine. We discuss the provision of an implementation of our abstract machine definitions in the concluding chapter, together with a number of techniques for further improving the performance.

1.4 Thesis Aims

In summary, the main aim of this thesis is the provision of a detailed definition of a runtime code-replacement strategy for the Standard ML programming language. Current techniques for runtime code-replacement, such as dynamic-linking and binary-compatibility, enable only the replacement of functions and do not provide any guarantees of safety for the replacement. By contrast, our replacement strategy will be a module-level operation over functions, types, and data. We will permit the complete replacement of one module (and its data) with another at runtime, under the condition that both modules match the same signature. We will also allow the extension of signatures, facilitating the introduction of new types and functions into the computation. In essence, any kind of replacement that can be expressed as a Standard ML type transformation can be performed. For example, data structures can be completely changed, functions can be optimised or rewritten, and mistakes in the source program can be corrected. Thus, our operation will accommodate a far greater degree of flexibility in performing runtime customisation and optimisation than is currently available. Our intention is that this should be achieved with only minor changes to the syntax and semantics of Standard ML, meaning that existing programs can be easily adapted for code-replacement.

The published semantics of Standard ML do not allow a definition of our intended model of code-replacement to be easily defined. In particular, the presence of type inference in the static semantics would unnecessarily complicate the definition, and the absence of memory management in the dynamic semantics means that our chosen technique cannot readily be defined. Consequently, we will take a layered approach to our definition, making use of intermediate representations, as in a typical compilation of the language. We will therefore define an explicitly typed intermediate language, called $\mathcal{M}\Lambda$, with an abstract machine semantics for providing code-replacement. We will then define a translation from the full language into $\mathcal{M}\Lambda$. We believe that there is significant novelty involved in this approach, e.g. modules translation and efficient tag-free garbage-collection, to make this a profitable exercise. In defining these stages we will take advantage of state-of-the-art advances in the fields of types in compilation and abstract machines. Furthermore, this layered approach mirrors all the stages that would be required in a compiler implementation of code-replacement.

In the latter part of the thesis we recognise the evolving nature of computer architectures, in particular the recent trend towards distributed execution of computer programs. Our aim here is to demonstrate that our code replacement operation is not limited to single machine architectures with a single thread of execution. We will first define an extension of the Dynamic ML programming language which facilitates a form of distributed evaluation. We will then show how our code replacement operations can be extended to cover this extended case. Distributed algorithms are notoriously difficult to construct correctly owing to unexpected interactions between asynchronous components. Consequently, we aim to gain confidence in our distributed extension by performing a mechanical verification of our definition using model-checking techniques.

The outline of work for the remainder of the thesis is as follows. In Chapters 2 through 7 we define the syntax and semantics of our Dynamic ML language, with procedures for dynamic compilation and replacement of modules. In Chapter 8 we extend the Dynamic ML language with distributed execution and in Chapter 9 adapt our models of dynamic compilation and replacement accordingly. In Chapter 10 we address the issue of verifying our code replacement algorithms. This is performed mechanically using a model checking technique. Finally, in Chapter 11 we summarise our results, discuss the extent to which the thesis aims have been achieved, and identify a number of directions for further research.

Chapter 2

$\mathcal{M}\Lambda$ Language

In this chapter we define the syntax of an explicitly typed intermediate language, called $\mathcal{M}\Lambda$, which we have developed for describing code replacement in Dynamic ML. The language is explicitly typed in that all type information is present in the syntax of the language. It is worth bearing in mind that programs in $\mathcal{M}\Lambda$ would not generally be constructed by hand, rather an $\mathcal{M}\Lambda$ program would be generated automatically by a translation operation. Explicit typing in an intermediate language consigns the complex problems associated with type inference completely to the front-end. The Dynamic ML language is a (syntactically) simplified variant of Standard ML. In Chapter 4 we present the syntax of Dynamic ML, and define a translation from Dynamic ML to $\mathcal{M}\Lambda$.

Of particular importance in our intermediate language is the representation of polymorphism. Therefore, at its lowest level, our $\mathcal{M}\Lambda$ language is based on a call-by-value variant of the Girard-Reynolds second-order polymorphic λ-calculus [Rey74], called System F_2. This calculus provides a general kind of polymorphism sufficient for our intermediate language. However, while System F_2 has significant theoretical importance, it would be very difficult to translate a real programming language into this calculus. Thus, our language is essentially a heavily sugared variant of the calculus. The relationship between $\mathcal{M}\Lambda$ and System F_2 is detailed in Chapter 3. Our language contains additional support for datatype declarations, let/letrec polymorphism, reference types, threads, and exceptions. The language is designed to be sufficiently rich to efficiently represent the entire core Standard ML language (and hence Dynamic ML). We are confident in this claim as the language has been adapted from an intermediate representation used in the ML-Kit with Regions compiler [TBE+98]. Despite the richness of our $\mathcal{M}\Lambda$ language, it is still at an altogether lower-level than Standard ML. Therefore, to aid the reader in understanding the language, we present many examples in this chapter comparing the syntax of Standard ML and $\mathcal{M}\Lambda$.

2.1 Notation and Mathematical Conventions

Before describing the syntax and semantics of the $\mathcal{M}\Lambda$ language, we briefly introduce the mathematical notation that will be used throughout the thesis. Nothing more complex than a basic knowledge of set theory will be required.

A *set* is defined by enumerating its members in braces, e.g:

$$\overline{\overline{x}} = \{\, red,\ orange,\ yellow,\ green,\ blue,\ violet \,\}$$

As a special case, $\{\}$ is the empty set, more commonly written $\emptyset$. The standard operations will be applied to sets: membership $\in$, union $\cup$, disjoint union $\uplus$, intersection $\cap$, and difference $\setminus$.

Given a set $\overline{\overline{x}}$, a *sequence* is an ordered list of members of $\overline{\overline{x}}$. Sequences are enclosed by brackets, e.g:

$$\overline{x} = (\, red,\ green,\ blue,\ red \,)$$

Unlike sets, sequences are ordered: the sequence $(\, red,\ blue \,)$ is distinct from the sequence $(\, blue,\ red \,)$. As indicated above, the same object may appear more than once in a sequence. Only *non-empty finite* sequences will be considered here. A sequence with k elements is written $\overline{x}^k$, where $k > 0$. The ith element of a sequence is x^i, where $0 < i \leq k$.

Given two sets $\overline{\overline{x}}$ and $\overline{\overline{y}}$, a *relation* is an arbitrary set of pairs $(\, x_i,\ y_i \,)$, where every x_i is a member of $\overline{\overline{x}}$, and every y_i is a member of $\overline{\overline{y}}$, e.g:

$$\mathcal{R} = \{\, (x_1,\ y_1),\ (x_2,\ y_2),\ (x_3,\ y_3),\ \ldots \,\}$$

The *domain* and *range* of a relation, written Dom $\mathcal{R}$ and Rng $\mathcal{R}$, are the sets of objects which appear as the first and second elements of the pairs respectively, for a *total* relation, the domain is the set $\overline{\overline{x}}$. Only finite partial functional relations, called *finite maps*, will be considered here. That is, relations with a finite domain, where there is at most one element of the range $\overline{\overline{y}}$ for each element of the domain $\overline{\overline{x}}$ in the relation. For simplicity, finite maps will be constructed as a one-to-one relation between sequences of equal length (the first sequence must have no repeated elements). The finite map from the sequence $\overline{x}^k$ to the sequence $\overline{y}^k$ is defined:

$$x \overset{map}{\mapsto} y = \{\, x^1 \ \mapsto\ y^1,\ \ldots,\ x^k \ \mapsto\ y^k \,\}$$

The pairs of elements in a finite map are written $x^i \ \mapsto\ y^i$ for clarity.

2.2 Abstract Syntax

The abstract syntax of the $\mathcal{M}\Lambda$ language is defined by the grammar presented in Figure 2.2. The $\mathcal{M}\Lambda$ language is composed of a *type language* and an *expression language*. The syntactic categories of the language include special constants *scon* (unit, integer, word, real, character, and string), user-defined type-names *tn*, and constructors *con*. Type names, constructor names, and exception names must be distinct and uniquely defined. For clarity, we will prefix type names with 't_' (e.g. t_bool and t_list), constructors with 'c_' (e.g. c_cons and c_true), and exceptions with 'e_' (e.g. e_overflow and e_match) in our examples. The meta-variable c ranges over special constants and constructors, and the meta-variable i over special constants of integer type. A number of functions are predefined on the special constants, as illustrated in Figure 2.1. These built-in functions coincide with the initial Basis of the Standard ML language [MTHM97].

Standard ML Operation	$\mathcal{M}\Lambda$ Function
Absolute Value (`abs`)	ABS_i, ABS_r
Negate (`~`)	NEG_i, NEG_r
Integer Divide (`div`)	DIV_i, DIV_w
Modulo (`mod`)	MOD_i, MOD_w
Divide (`/`)	DIV_r
Multiply (`*`)	MUL_i, MUL_w, MUL_r
Add (`+`)	ADD_i, ADD_w, ADD_r
Subtract (`-`)	SUB_i, SUB_w, SUB_r
Less (`<`)	LT_i, LT_w, LT_r, LT_c, LT_s
Greater (`>`)	GT_i, GT_w, GT_r, GT_c, GT_s
Less or Equal (`<=`)	LEQ_i, LEQ_w, LEQ_r, LEQ_c, LEQ_s
Greater or Equal (`>=`)	GEQ_i, GEQ_w, GEQ_r, GEQ_c, GEQ_s
Equal (`=`)	EQ_i, EQ_w, EQ_r, EQ_c, EQ_s

i = integer, w = word, r = real, c = char, s = string

Figure 2.1: *Built-in Functions.*

There are two classes of variables in the language: *lambda variables* x, and *type variables* α. Lambda variables, corresponding to variables in the lambda calculus, are bound to values generated by the evaluation of expressions, and type variables α are bound to types. Both lambda variables and type variables must also be uniquely defined in $\mathcal{M}\Lambda$. The uniqueness property means that we do not have to worry about renaming (α-conversion) during function applications. Therefore, during translation into $\mathcal{M}\Lambda$, a renaming operation on the variables will generally be required.

$\tau \in$ Type	$::=$	tn	(type name)
	$\mid$	$tn(\overline{\tau}^k)$	(constructed type)
	$\mid$	$\overline{\tau}^k$	(tuple type)
	$\mid$	$\tau_1 \rightarrow \tau_2$	(function type)
	$\mid$	α	(type variable)
$\sigma \in$ Type scheme	$::=$	τ	(monotype)
	$\mid$	$\forall\, \overline{\alpha}^k .\, \tau$	(polytype)
$P \in$ Program	$::=$	$(\overline{\overline{D}},\ \overline{\overline{X}},\ E)$	
$D \in$ Datatype	$::=$	$\mathbf{datatype}\ tn\ \mathbf{of}\ \overline{\overline{C}}$	(monomorphic)
	$\mid$	$\mathbf{datatype}\ (\overline{\alpha}^k,\ tn)\ \mathbf{of}\ \overline{\overline{C}}$	(polymorphic)
$X \in$ Exception	$::=$	$\mathbf{exception}\ C$	
$C \in$ Constructor	$::=$	con	(nullary declaration)
	$\mid$	$(con,\ \tau)$	(unary declaration)
$E \in$ Expression	$::=$	$\mathbf{scon}\ scon$	(special constant)
	$\mid$	$\mathbf{var}\ x$	(variable)
	$\mid$	$\mathbf{var}\ (x,\ \overline{\tau}^k)$	(polymorphic var)
	$\mid$	$\mathbf{fn}\ (x,\ \tau_1 \rightarrow \tau_2)\ =\ E$	(function)
	$\mid$	$\mathbf{fn}\ (\overline{x}^k,\ \overline{\tau_1}^k \rightarrow \tau_2)\ =\ E$	(multi-argument fn)
	$\mid$	$\mathbf{con}\ con$	(nullary constructor)
	$\mid$	$\mathbf{con}\ (con,\ E)$	(unary constructor)
	$\mid$	$\mathbf{con}\ (con,\ \overline{\tau}^k)$	(nullary polymorphic)
	$\mid$	$\mathbf{con}\ (con,\ \overline{\tau}^k,\ E)$	(unary polymorphic)
	$\mid$	$\mathbf{decon}\ (con,\ E)$	(deconstructor)
	$\mid$	$\mathbf{decon}\ (con,\ \overline{\tau}^k,\ E)$	(polymorphic decon)
	$\mid$	$\mathbf{tuple}\ \overline{E}^k$	(tuple)
	$\mid$	$\mathbf{select}\ (i,\ E)$	(selection)
	$\mid$	$\mathbf{let}\ (x,\ \sigma)\ =\ E_1\ \mathbf{in}\ E_2$	(binding)
	$\mid$	$\mathbf{let}\ \overline{(x,\ \sigma)}^k\ =\ E_1\ \mathbf{in}\ E_2$	(multi-binding)
	$\mid$	$\mathbf{switch}\ E_1\ \mathbf{case}\ (c\ \overset{map}{\mapsto}\ E_2,\ E_3)$	(conditional)
	$\mid$	$\mathbf{fix}\ \overline{(x,\ \sigma)\ =\ E_1}^k\ \mathbf{in}\ E_2$	(recursive binding)
	$\mid$	$\mathbf{app}\ (E_1,\ E_2)$	(application)
	$\mid$	$\mathbf{app}\ (E_1,\ \overline{E_2}^k)$	(multi-argument app)
	$\mid$	$\mathbf{assign}\ (E_1,\ E_2)$	(assignment)
	$\mid$	$\mathbf{raise}\ (E,\ \tau)$	(raise exception)
	$\mid$	$\mathbf{handle}\ E_1\ \mathbf{with}\ E_2$	(handle exception)

Figure 2.2: $\mathcal{M}\Lambda$ *Abstract Syntax.*

2.2.1 Type Language

The type language is structured into two levels: a type level τ, and a type scheme level σ. The type level defines the monomorphic types, usually called monotypes. These include user-defined type names *tn* and $tn(\overline{\tau}^k)$. The second form is used for representing instances of polymorphic datatypes e.g. t_list(t_int). The type names include the basic types, as required by the special constants; value constructor types; reference types; and exception types. Function values have function types $\tau_1 \rightarrow \tau_2$. For convenience, we also include a tuple type $\overline{\tau}^k$ which can be used to type records and functions with more than one argument. Finally, we have type variables α.

Polymorphic types, called polytypes, are defined by type-schemes σ. At this level we have the universally quantified type $\forall\ \overline{\alpha}^k.\ \tau$, where $\overline{\alpha}^k$ is a sequence of type variables that occur in τ. Since we do not allow empty sequences, we include the monotype τ in the definition. We say that a type variable in τ is *bound* if it is quantified by the polytype, i.e. it appears in the sequence $\forall\ \overline{\alpha}^k$, and *free* otherwise. A *closed* polytype is one with no free type variables. As an example, the polymorphic identity function is represented by the (closed) type scheme $\forall \alpha\ .\ \alpha \rightarrow \alpha$.

2.2.2 Expression Language

The expression language is structured as a hierarchy; the top of which is a program P. A program contains a set of datatype declarations $\overline{\overline{D}}$, a set of exception declarations $\overline{\overline{X}}$, and a single expression E. Each datatype declaration consists of a unique type name *tn* and a set of constructors $\overline{\overline{C}}$. Datatypes may be polymorphic, in which case the declaration includes a list of type variables $\overline{\alpha}^k$. Each constructor declaration may be nullary *con* or unary (*con*, τ). As an example, a polymorphic list datatype, containing a nullary and unary constructor, is represented in $\mathcal{M}\Lambda$ according to Figure 2.3. The remaining $\mathcal{M}\Lambda$ expressions E divide into the following categories: values, value constructors, structured expressions, function expressions, references, and exceptions.

Standard ML:
```
datatype 'a list = nil
                 | cons of 'a * 'a list
```

$\mathcal{M}\Lambda$ Syntax: **datatype** ((α), t_list) **of** {c_nil, (c_cons, (α, t_list(α)))}

Figure 2.3: *Datatype Example.*

Values

The $\mathcal{M}\Lambda$ values are the special constants, e.g. **scon** 7, variables, and functions. Variables are referenced by **var** expressions. If the variable is polymorphic, an instance may be created by supplying types $\overline{\tau}^k$. Functions values (λ-abstractions) are created by **fn** expressions as shown in Figure 2.4. A function may take a single argument or a tuple of arguments, allowing both curried and uncurried forms to be represented.

Standard ML:

```
fn (x : int) => fn y => x + y
fn (x : int, y) => x + y
```

$\mathcal{M}\Lambda$ Syntax:

$\textbf{fn}\ (\text{x},\ \text{t\_int} \rightarrow (\text{t\_int} \rightarrow \text{t\_int}))\ =$
$\quad \textbf{fn}\ (\text{y},\ \text{t\_int} \rightarrow \text{t\_int})\ =\ \textbf{app}\ (\text{ADD}_i,\ \textbf{tuple}\ (\textbf{var}\ \text{x},\ \textbf{var}\ \text{y}))$
$\textbf{fn}\ ((\text{x},\ \text{y}),\ (\text{t\_int},\ \text{t\_int}) \rightarrow \text{t\_int})\ =$
$\quad \textbf{app}\ (\text{ADD}_i,\ \textbf{tuple}\ (\textbf{var}\ \text{x},\ \textbf{var}\ \text{y}))$

Figure 2.4: *Function Example.*

Value Constructors

The $\mathcal{M}\Lambda$ value constructors are used to generate datatype values. There are four forms of **con** expression corresponding to nullary and unary constructors for both monomorphic and polymorphic datatypes respectively. Deconstructor expressions **decon** are also provided for the unary constructors. Deconstructors are generally used for representing pattern matching operations. An example illustrating the usage of these expressions is given in Figure 2.5.

Standard ML:

```
datatype bool = true | false;
val x = true;
val y = [1];
fn (h :: t) => h;
```

$\mathcal{M}\Lambda$ Syntax:

$\textbf{datatype}\ \text{t\_bool}\ \textbf{of}\ \{\text{c\_true},\ \text{c\_false}\}$
$\textbf{let}\ (\text{x},\ \text{t\_bool})\ =\ \textbf{con}\ \text{c\_true}$
$\textbf{let}\ (\text{y},\ \text{t\_list}(\text{t\_int}))\ =$
$\quad \textbf{con}\ (\text{c\_cons},\ (\text{t\_int}),\ \textbf{tuple}\ (\textbf{scon}\ 1,\ \textbf{con}\ (\text{c\_nil},\ (\text{t\_int}))))$
$\textbf{fn}\ (\text{l},\ \text{t\_list}(\alpha) \rightarrow \alpha)\ =$
$\quad \textbf{let}\ ((\text{h},\ \text{t}),\ (\alpha,\ \text{t\_list}(\alpha)))\ =\ \textbf{decon}\ (\text{c\_list},\ \textbf{var}\ \text{l})$
$\quad \textbf{in}\ \textbf{var}\ \text{h}$

Figure 2.5: *Constructor Example.*

Structured Expressions

The **let** expression binds values to lambda variables. Figure 2.6 illustrates a renaming of Standard ML variables to ensure the uniqueness of the identifiers.

Standard ML:

```
let val y = 2
    val y = y + y
in y end;
```

$\mathcal{M}\Lambda$ Syntax:

let (a, t_int) = **scon** 2
in let (b, t_int) = **app** (ADD$_i$, **tuple** (**var** a, **var** a))
  in var b

Figure 2.6: *Variable Renaming Example.*

Standard ML records and tuples are represented in $\mathcal{M}\Lambda$ by **tuple** expressions. During translation from Standard ML to $\mathcal{M}\Lambda$, the field labels of a record are sorted alphabetically. This avoids the need to retain these labels in $\mathcal{M}\Lambda$. The fields of a tuple are accessed using **select** and **let** expressions (Figure 2.7).

Standard ML:

```
let val file = {size = 17373, name = "thesis.tex"}
in #size file end;
let val vector = (1.0, 1.5, 6.8)
    val (x, y, z) = vector
in x end;
```

$\mathcal{M}\Lambda$ Syntax:

let (file, (t_string, t_int)) =
  tuple (**scon** "thesis.tex", **scon** 17373)
in select (2, **var** file)
let (vector, (t_real, t_real, t_real)) =
  tuple (**scon** 1.0, **scon** 1.5, **scon** 6.8)
in let ((x, y, z), (t_real, t_real, t_real)) = **var** vector
  in var x

Figure 2.7: *Records and Tuples Example.*

The **switch** expression is used to perform control flow. In particular, pattern matching is represented by switch statements in $\mathcal{M}\Lambda$. A **switch** statement contains an argument expression E_1, a finite map from constants c to case expressions E_2, and a wild-card expression E_3. If the argument expression evaluates to a constant which is in the domain of the finite map, the corresponding expression from the range is subsequently evaluated. If there is no matching constant in the domain, the wild-card expression is evaluated.

Function Expressions

The **let** expression may be used to bind non-recursive function values to lambda variables. However, recursive and mutually-recursive functions must be bound using the fixed-point expression (often called 'letrec' in other lambda languages). The **fix** expression binds a sequence of functions values to lambda variables simultaneously, such that any function may refer to any other.

Function application is performed by an **app** expression which applies the function E_1 to the argument E_2 (or arguments $\overline{E_2}^k$). Function application corresponds to β-reduction in the λ-calculus. As we will see in Chapter 5, a *call-by-value* strategy is used in $\mathcal{M}\Lambda$ for function application, meaning that E_1 and E_2 are fully evaluated before the function is applied. For convenience, there are two forms of **app** expression which correspond to the single and multiple argument forms of function declaration.

An example is given in Figure 2.8 to illustrate the representation of pattern matching using a switch expression, the use of a fixed-point expression for declaring a recursive function, and the use of application expressions. Note that this example is a complete $\mathcal{M}\Lambda$ program, rather than a fragment of $\mathcal{M}\Lambda$ expressions as previously illustrated in our examples. Also note that the wild-card of the **switch** expression is defined to raise a match exception since there are no other appropriate values. Exceptions are described in the following section.

Standard ML:

```
fun fact 0 = 1
  | fact n = n * fact(n - 1);
fact 10;
```

$\mathcal{M}\Lambda$ Syntax:

```
(∅, ∅,
  fix ((fact, t_int → t_int) =
    fn (n, t_int → t_int) =
      switch (app (EQ_i, tuple (var n, scon 0)))
      case ({c_true ↦ scon 1,
              c_false ↦
                app (MUL_i, tuple (var n,
                  app (var fact, app (SUB_i, tuple (var n, scon 1)))))},
      raise (con e_match, t_int)))
  in app (var fact, scon 10))
```

Figure 2.8: *Factorial Example.*

References and Exceptions

References and exceptions are represented in $\mathcal{M}\Lambda$ by special reserved datatypes t_ref and t_exn respectively. These datatypes would be defined in Standard ML according to Figure 2.9.

```
datatype 'a t_ref = c_ref of 'a

datatype t_exn = e_match
               | e_bind
               | e_overflow
               | ...
               | e_error of int * string
               | ...
```

Figure 2.9: *Reference and Exception Datatypes.*

References are represented as *values* of the polymorphic constructor c_ref. References are constructed and deconstructed using the polymorphic forms of the **con** and **decon** expressions described earlier. A special **assign** $(E_1,\ E_2)$ expression is provided for updating the value of a reference, where E_1 evaluates to a reference, and E_2 evaluates to the new value. An example illustrating the usage of references in $\mathcal{M}\Lambda$ is given in Figure 2.10. The dereferencing function *!* is part of the Standard ML Basis.

Standard ML:

```
let fun ! (ref x) = x
    val a = ref 6
    val _ = a := 9
    val b = !a
in b end;
```

$\mathcal{M}\Lambda$ Syntax:

```
(∅, ∅,
  let (deref, t_ref(α) → α) =
    fn (x, t_ref(α) → α) =
      decon (c_ref, var x)
  in let (a, t_ref(t_int)) =
      con (c_ref, (t_int), scon 6)
     in let (_, t_unit) =
         assign (var a, scon 9)
       in let (b, t_int) =
           app (var (deref, (t_int)), var a)
         in var b)
```

Figure 2.10: *References Example.*

By contrast, exceptions are represented as *constructors* of the monomorphic datatype t_exn. Each **exception** declaration appears as a separate constructor of t_exn. In $\mathcal{M}\Lambda$, exceptions are statically declared at the beginning of a program. This avoids the need to deal with a datatype which has a varying number of constructors. The exceptions e_match, e_bind, and e_overflow are predefined in $\mathcal{M}\Lambda$. As in Standard ML we do not allow polymorphic exceptions.

We construct exceptions, called *exception packets*, using the monomorphic forms of the **con** expressions. Exception packets can be deconstructed using the **decon** expression. The **raise** $(E,\ \tau)$ expression is used to raise an exception, where E evaluates to an exception packet, and τ corresponds to the type that would be required if an exception were not raised. A **raise** expression causes a change in the flow of control. The **handle** expression tests whether E_1 raises an exception packet. If so, the function expression E_2 is applied to the packet. An example, which uses exceptions to prevent underflow of the factorial function, from Figure 2.8, is presented in Figure 2.11.

Standard ML:

```
exception Factorial
fun errorfact n = if n < 0 then raise Factorial
                  else fact n
fun nicefact n = errorfact n handle Factorial => 0
nicefact ~10;
```

$\mathcal{M}\Lambda$ Syntax:

```
(∅, {exception e_factorial},
  let (errorfact, t_int → t_int) =
    fn (n, t_int → t_int) =
      switch (app (LT_i, tuple (var n, scon 0)))
      case ({c_true ↦ raise (con e_factorial, t_int),
             c_false ↦ app (var fact, var n)},
      raise (con e_match, t_int))
  in let (nicefact, t_int → t_int) =
    fn (n, t_int → t_int) =
      handle (app (var errorfact, var n))
      with
        fn (exn, t_exn → t_int) =
          switch (var exn)
          case ({e_factorial ↦ scon 0}, raise (var exn, t_int))
    in app (var fac, scon −10))
```

Figure 2.11: *Exceptions Example.*

2.3 Threaded Concurrency

The $\mathcal{M}\Lambda$ language also contains expressions for creating and communicating between *threads*. These threads are not part of Standard ML, rather they are derived from Concurrent ML, a distributed extension described in [Mat91]. This extension is designed for executing ML programs on a distributed network of workstations. The Concurrent ML language extends Standard ML with the language constructs illustrated in Figure 2.12.

```
eqtype α channel
val channel     :  unit → α channel
val send        :  α channel × α → unit
val receive     :  α channel → α
val fork        :  (unit → unit) → unit
val rfork       :  int × (unit → unit) → unit
```

Figure 2.12: *Concurrent ML Language Extensions.*

Communication between threads is performed via blocking polymorphic channels of type α `channel`. The type system ensures that values sent and received on a particular channel have the same type. New channels are constructed using the ***channel*** function, and may be passed around as values. A thread sends a value to another thread using the ***send*** function, which takes two arguments: a channel and a value. The receiving thread uses the ***receive*** function, which takes a channel argument. Both threads are blocked until the value is passed, which happens atomically. Threads are created either using the ***fork*** function, which runs the child thread on the same processor as its parent, or the ***rfork*** function, which runs the child thread on a specified processor. Both ***fork*** and ***rfork*** take a function argument, whose body is evaluated in the child thread. When the function returns, the thread is terminated. The ***rfork*** function takes an additional integer argument which specifies where the child thread is to run.

As with references, we represent channels in $\mathcal{M}\Lambda$ as values of the polymorphic constructor c_chan of a reserved datatype t_chan. In Concurrent ML the communication and threading primitives are represented by functions. However, applying one of these functions has the side-effect of sending a value, creating a thread, etc. Since we cannot represent these operations by the usual functions in $\mathcal{M}\Lambda$, we include extra expressions in the language which correspond to these operations. The t_chan datatype, and the extra expressions are illustrated in Figure 2.13.

$$\textbf{datatype}\ ((\alpha),\ \text{t\_chan})\ \textbf{of}\ \{(\text{c\_chan},\ \alpha)\} \qquad \text{(channel datatype)}$$

$$\begin{array}{lcll}
E \in \text{Expression} & ::= & \textbf{fork}\ E & \text{(local thread)} \\
 & | & \textbf{rfork}\ (E_1,\ E_2) & \text{(remote thread)} \\
 & | & \textbf{send}\ (E_1,\ E_2) & \text{(send value)} \\
 & | & \textbf{receive}\ E & \text{(receive value)}
\end{array}$$

Figure 2.13: $\mathcal{M}\Lambda$ *Thread Expressions.*

An example illustrating the thread operations in Concurrent ML, and their translation into $\mathcal{M}\Lambda$ is shown in Figure 2.14. In this example, a thread is created on processor 2 that simply waits for an integer value on channel `c`, then returns the value, incremented by 3, on the same channel. The example concludes by sending the value 7 to the thread, and returns the response (which will be 10).

Concurrent ML:

```
let val c = channel()
    val _ = rfork(2,
                  fn _ => send(c, receive(c) + 3))
    val _ = send(c, 7)
in
    receive(c)
end
```

$\mathcal{M}\Lambda$ **Syntax:**

$$\begin{array}{l}
(\emptyset,\ \emptyset,\ \textbf{let}\ (c,\ \text{t\_chan}(\text{t\_int}))\ =\ \textbf{con}(\text{c\_chan},\ (\text{t\_int})) \\
\quad \textbf{in let}\ (\_,\ \text{t\_unit})\ =\ \textbf{rfork}\ (\textbf{scon}\ 2, \\
\qquad\quad \textbf{fn}\ (\_,\ \text{t\_unit} \rightarrow \text{t\_unit})\ = \\
\qquad\qquad \textbf{send}\ (\textbf{var}\ c, \\
\qquad\qquad\quad \textbf{app}\ (\text{ADD}_i,\ \textbf{tuple}\ (\textbf{receive}\ (\textbf{var}\ c),\ \textbf{scon}\ 3)))) \\
\qquad \textbf{in let}\ (\_,\ \text{t\_unit})\ =\ \textbf{send}\ (\textbf{var}\ c,\ \textbf{scon}\ 7) \\
\qquad\quad \textbf{in receive}\ (\text{var}\ c))
\end{array}$$

Figure 2.14: *Threads Example.*

Chapter 3

Static Semantics of $\mathcal{M}\Lambda$

We define the static semantics of the $\mathcal{M}\Lambda$ intermediate language in this chapter by presenting a formal *type system*. The type system is defined by a collection of rules which are used to determine if a program is *well-typed*. Well-typing corresponds to a notion of predictability of program behaviour. The syntax of $\mathcal{M}\Lambda$ is based on an enriched variant of System F_2. It therefore follows that our type system is derived from the type system of System F_2. We discuss the relation between our type system and the type system for System F_2 at the end of this chapter.

$\Gamma \vdash \diamond$	(well-formedness)
$\Gamma \vdash \sigma$	(existence)
$\Gamma \vdash \sigma_1 = \sigma_2$	(equivalence)
$\Gamma \vdash P : \tau$	(typing)

Figure 3.1: *Type System Judgements.*

The description of a type system begins with a collection of rules called *judgements*. The judgements for the $\mathcal{M}\Lambda$ language are given in Figure 3.1. Judgements are formulated as *assertions*. The well-formedness judgement asserts that an environment Γ is *well formed*. The existence judgement asserts that σ is a well-formed type-scheme in Γ. The equivalence judgement asserts that the type-schemes σ_1 and σ_2 are equivalent in Γ. The typing judgement asserts that P is a well-formed program with type τ in Γ. The rules in the remainder of this chapter assert the *validity* of these judgements. Validity of the judgements formalises well-typing. Validity is derived inductively from judgements that are already known to be valid. The process begins with the intrinsically valid judgement $\emptyset \vdash \diamond$ which asserts that the empty environment is valid.

3.1 Static Typing-Environments

Our formal judgements all refer to a static typing environment Γ. We must therefore provide a definition for this environment before proceeding. The main purpose of the environment is to track the allocation of the variables and their corresponding types. In $\mathcal{M}\Lambda$ we also allow variables of user-defined types. Thus, our environment must track these also. The structure of the $\mathcal{M}\Lambda$ static typing-environment is illustrated in Figure 3.2.

$$
\begin{array}{llll}
\text{Environment} & \Gamma & ::= & (\mathit{TE},\ \mathit{VE},\ \mathit{DE},\ \mathit{CE}) \\
 & & & \\
\text{Type Variables} & \mathit{TE} & ::= & \overline{\overline{\alpha}} \\
\text{Lambda Variables} & \mathit{VE} & ::= & x \overset{map}{\longmapsto} \sigma \\
\text{Datatype Names} & \mathit{DE} & ::= & \mathit{tn} \overset{map}{\longmapsto} (n,\ \overline{\overline{\mathit{con}}}) \\
\text{Constructors} & \mathit{CE} & ::= & \mathit{con} \overset{map}{\longmapsto} \sigma
\end{array}
$$

Figure 3.2: $\mathcal{M}\Lambda$ *Static Typing Environment.*

The environment is represented by a four-tuple comprising a type-variable environment, a lambda-variable environment, a datatype environment, and a constructor environment. The type-variable environment *TE* maintains a set of type variables. This set of type variables is used to determine the validity of a type scheme (Section 3.2). The lambda-variable environment *VE* maps bound variable identifiers to type schemes. We can use a finite map because the variable identifiers are unique. The datatype environment *DE* maps the type names *tn* to sets of constructors $\overline{\overline{\mathit{con}}}$. The number of free type variables in the datatype (called the *arity*) is given by the parameter n. Finally, the constructor environment *CE* maps constructors to type schemes.

We will use the notation $\Gamma[\theta]$ for extending an environment Γ with a binding θ, where θ is one of α, $x \mapsto \sigma$, $\mathit{tn} \mapsto (n,\ \overline{\overline{\mathit{con}}})$, or $\mathit{con} \mapsto \sigma$. We do not need to specify the sub-environment in Γ as it follows from the naming of θ; type variables are named α; type names are prefixed by t_; constructors are prefixed by c_; exceptions are prefixed e_; anything else is considered a lambda variable. We will use the notation $\Gamma[\theta^1,\ \ldots,\ \theta^k]$ for adding multiple binding of the same kind to an environment, and $\Gamma[\theta_1]\cdots[\theta_k]$ for adding multiple bindings of different kinds. The notation $\Gamma(\phi)$ will be used for retrieving a binding ϕ from an environment Γ, where ϕ is one of α, x, tn, or con. For simplicity, we will define the domain of the environment Dom Γ as the union of the domains of all the finite-maps in the environment, together with the set of type variables.

3.1.1 The Initial Environment

In the previous chapter we saw that the $\mathcal{M}\Lambda$ language contains a number of predefined functions and types, corresponding to the initial static basis of Standard ML [MTHM97]. These functions, and their types, are included in the initial typing environment of $\mathcal{M}\Lambda$, shown in Figure 3.3. The initial lambda variable environment contains the primitive functions on the special constants. For brevity, only the operations on the integers are shown.

The initial types include the basic types for the special constants: t_unit, t_int, t_word, t_real, t_char, and t_string. These types appear with an arity of 0 (i.e. they are non polymorphic) and are defined with an empty set of constructors. The definition of the boolean datatype t_bool references the two nullary constructors c_true and c_false. The type of a nullary constructor is simply the parent datatype, in this case t_bool. The definition of the list datatype references a nullary constructor c_nil and a unary constructor c_cons. The type of a unary constructor is a function type from the arguments of the constructor to the parent datatype, in this case $\forall(\alpha).\ (\alpha,\ \text{t\_list}(\alpha)) \rightarrow \text{t\_list}(\alpha)$.

$$\Gamma \quad = \quad (\emptyset,\ VE,\ DE,\ CE)$$

$$VE \quad = \quad \left\{ \begin{array}{l} \text{ABS}_i \ \mapsto\ \text{t\_int} \rightarrow \text{t\_int},\ \text{NEG}_i \ \mapsto\ \text{t\_int} \rightarrow \text{t\_int}, \\ \text{DIV}_i \ \mapsto\ (\text{t\_int},\ \text{t\_int}) \rightarrow \text{t\_int},\ \text{MOD}_i \ \mapsto\ (\text{t\_int},\ \text{t\_int}) \rightarrow \text{t\_int}, \\ \text{MUL}_i \ \mapsto\ (\text{t\_int},\ \text{t\_int}) \rightarrow \text{t\_int}, \\ \text{ADD}_i \ \mapsto\ (\text{t\_int},\ \text{t\_int}) \rightarrow \text{t\_int},\ \text{SUB}_i \ \mapsto\ (\text{t\_int},\ \text{t\_int}) \rightarrow \text{t\_int}, \\ \text{LT}_i \ \mapsto\ (\text{t\_int},\ \text{t\_int}) \rightarrow \text{t\_bool},\ \text{GT}_i \ \mapsto\ (\text{t\_int},\ \text{t\_int}) \rightarrow \text{t\_bool}, \\ \text{LEQ}_i \ \mapsto\ (\text{t\_int},\ \text{t\_int}) \rightarrow \text{t\_bool},\ \text{GEQ}_i \ \mapsto\ (\text{t\_int},\ \text{t\_int}) \rightarrow \text{t\_bool}, \\ \text{EQ}_i \ \mapsto\ (\text{t\_int},\ \text{t\_int}) \rightarrow \text{t\_bool}, \end{array} \right\}$$

$$DE \quad = \quad \left\{ \begin{array}{l} \text{t\_unit} \ \mapsto\ (0,\ \emptyset),\ \text{t\_int} \ \mapsto\ (0,\ \emptyset),\ \text{t\_word} \ \mapsto\ (0,\ \emptyset), \\ \text{t\_real} \ \mapsto\ (0,\ \emptyset),\ \text{t\_char} \ \mapsto\ (0,\ \emptyset),\ \text{t\_string} \ \mapsto\ (0,\ \emptyset), \\ \text{t\_bool} \ \mapsto\ (0,\ \{\text{c\_true},\ \text{c\_false}\}),\ \text{t\_list} \ \mapsto\ (1,\ \{\text{c\_nil},\ \text{c\_cons}\}), \\ \text{t\_ref} \ \mapsto\ (1,\ \{\text{c\_ref}\}),\ \text{t\_chan} \ \mapsto\ (1,\ \{\text{c\_chan}\}), \\ \text{t\_exn} \ \mapsto\ (0,\ \{\text{e\_div},\ \text{e\_mod},\ \text{e\_match},\ \text{e\_bind},\ \text{e\_overflow}\}) \end{array} \right\}$$

$$CE \quad = \quad \left\{ \begin{array}{l} \text{c\_true} \ \mapsto\ \text{t\_bool},\ \text{c\_false} \ \mapsto\ \text{t\_bool}, \\ \text{c\_nil} \ \mapsto\ \forall(\alpha).\ \text{t\_list}(\alpha),\ \text{c\_cons} \ \mapsto\ \forall(\alpha).\ (\alpha,\ \text{t\_list}(\alpha)) \rightarrow \text{t\_list}(\alpha), \\ \text{c\_ref} \ \mapsto\ \forall(\alpha).\ \alpha \rightarrow \text{t\_ref}(\alpha),\ \text{c\_chan} \ \mapsto\ \forall(\alpha).\ \alpha \rightarrow \text{t\_chan}(\alpha), \\ \text{e\_match} \ \mapsto\ \text{t\_exn},\ \text{e\_bind} \ \mapsto\ \text{t\_exn},\ \text{e\_overflow} \ \mapsto\ \text{t\_exn} \end{array} \right\}$$

Figure 3.3: *Initial Static Environment.*

3.1.2 Well-Formed Environments

Given an environment Γ, we can define the first judgement $\Gamma \vdash \diamond$, which asserts that Γ is a well-formed environment. Informally, well-formedness of an environment corresponds to the following conditions: all finite-maps are correctly constructed (e.g. no duplicates in the domain), all datatype constructors are defined, and any types which appear in the range are valid.

We define well-formedness inductively in Figure 3.4 with the base-case $\emptyset \vdash \diamond$ (Rule 3.1): the empty environment is well-formed. The remaining rules extend the environment Γ with a new entry and check for validity. For example, extending Γ with a new lambda variable: $\Gamma[x \;\mapsto\; \sigma]$ (Rule 3.3) is valid if σ is a valid type in Γ and x has not previously been declared. We must add a datatype and its constructors simultaneously (Rule 3.5) as a datatype refers to its constructors which in turn refer to the datatype. The *dtype* function (3.6) returns the parent datatype of a constructor. There are four cases corresponding to nullary and unary constructors of monomorphic and polymorphic datatypes.

$$\frac{}{\emptyset \vdash \diamond} \tag{3.1}$$

$$\frac{\Gamma \vdash \diamond \qquad \alpha \notin \mathrm{Dom}\,\Gamma}{\Gamma[\alpha] \vdash \diamond} \tag{3.2}$$

$$\frac{\Gamma \vdash \diamond \qquad \Gamma \vdash \sigma \qquad x \notin \mathrm{Dom}\,\Gamma}{\Gamma[x \;\mapsto\; \sigma] \vdash \diamond} \tag{3.3}$$

$$\frac{\Gamma \vdash \diamond \qquad tn \notin \mathrm{Dom}\,\Gamma}{\Gamma[tn \;\mapsto\; (0,\, \emptyset)] \vdash \diamond} \tag{3.4}$$

$$\frac{\begin{array}{l}\Gamma \vdash \diamond \qquad tn \notin \mathrm{Dom}\,\Gamma \qquad n \geq 0 \\ \overline{\overline{con_1}} = \{con_2^1,\, \ldots,\, con_2^k\} \qquad dtype(\sigma^1) = tn \quad \cdots \quad dtype(\sigma^k) = tn \\ \Gamma' = \Gamma[tn \;\mapsto\; (n,\, \overline{\overline{con_1}})] \qquad \Gamma' \vdash \sigma^1 \quad \cdots \quad \Gamma' \vdash \sigma^k\end{array}}{\Gamma[tn \;\mapsto\; (n,\, \overline{\overline{con_1}})][con_2^1 \;\mapsto\; \sigma^1,\, \ldots,\, con_2^k \;\mapsto\; \sigma^k] \vdash \diamond} \tag{3.5}$$

$$\begin{array}{lcl} dtype(tn) & = & tn \\ dtype(\tau \rightarrow tn) & = & tn \\ dtype(\forall\, \overline{\alpha}^k.\, tn(\overline{\alpha}^k)) & = & tn \\ dtype(\forall\, \overline{\alpha}^k.\, \tau \rightarrow tn(\overline{\alpha}^k)) & = & tn \end{array} \tag{3.6}$$

Figure 3.4: *Valid Environments.*

3.2 Type Schemes

We can now define the existence judgement $\Gamma \vdash \sigma$ and equivalence judgement $\Gamma \vdash \sigma_1 = \sigma_2$ over type-schemes (Figure 3.5). The environment Γ is assumed to be well-formed. The rules for existence (Rule 3.7 through 3.12) are largely trivial. For example, a function type is valid $\Gamma \vdash \tau_1 \to \tau_2$ (Rule 3.10) if the argument type is valid $\Gamma \vdash \tau_1$ and the result type is valid $\Gamma \vdash \tau_2$. For a polymorphic type-scheme $\Gamma \vdash \forall\ \overline{\alpha}^k.\ \tau$, the type variables $\overline{\alpha}^k$ are temporarily added to the environment while the type τ is checked for validity. The rules for equivalence (Rule 3.13 through 3.18) are similarly straightforward. Types are matched by virtue of being *structurally* equivalent. For a type-scheme (Rule 3.18), equivalence is checked by substituting the type variables for a common set $\overline{\alpha_3}^k$ and checking the resulting types for equivalence.

$$\frac{\Gamma(tn) = (0,\ \overline{\overline{con}})}{\Gamma \vdash tn} \quad (3.7)$$

$$\frac{\Gamma(tn) = (k,\ \overline{\overline{con}}) \qquad \Gamma \vdash \overline{\tau}^k}{\Gamma \vdash tn(\overline{\tau}^k)} \quad (3.8)$$

$$\frac{\Gamma \vdash \tau^1 \quad \cdots \quad \Gamma \vdash \tau^k}{\Gamma \vdash \overline{\tau}^k} \quad (3.9)$$

$$\frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2} \quad (3.10)$$

$$\frac{\alpha \in \mathrm{Dom}\,\Gamma}{\Gamma \vdash \alpha} \quad (3.11)$$

$$\frac{\Gamma[\alpha^1,\ \ldots,\ \alpha^k] \vdash \tau}{\Gamma \vdash \forall\ \overline{\alpha}^k.\ \tau} \quad (3.12)$$

$$\frac{tn \in \mathrm{Dom}\,\Gamma}{\Gamma \vdash tn = tn} \quad (3.13)$$

$$\frac{\Gamma \vdash tn_1 = tn_2 \qquad \Gamma \vdash \overline{\tau_1}^k = \overline{\tau_2}^k}{\Gamma \vdash tn_1(\overline{\tau_1}^k) = tn_2(\overline{\tau_2}^k)} \quad (3.14)$$

$$\frac{\Gamma \vdash \tau_1^1 = \tau_2^1 \quad \cdots \quad \Gamma \vdash \tau_1^k = \tau_2^k}{\Gamma \vdash \overline{\tau_1}^k = \overline{\tau_2}^k} \quad (3.15)$$

$$\frac{\Gamma \vdash \tau_1 = \tau_3 \qquad \Gamma \vdash \tau_2 = \tau_4}{\Gamma \vdash \tau_1 \to \tau_2 = \tau_3 \to \tau_4} \quad (3.16)$$

$$\frac{\alpha \in \mathrm{Dom}\,\Gamma}{\Gamma \vdash \alpha = \alpha} \quad (3.17)$$

$$\frac{\begin{array}{l}\overline{\alpha_3}^k \text{ fresh}\\ \mathcal{S}_1 = \{\alpha_1^1 \mapsto \alpha_3^1,\ \ldots,\ \alpha_1^k \mapsto \alpha_3^k\}\\ \mathcal{S}_2 = \{\alpha_2^1 \mapsto \alpha_3^1,\ \ldots,\ \alpha_2^k \mapsto \alpha_3^k\}\\ subst(\mathcal{S}_1,\ \tau_1) = subst(\mathcal{S}_2,\ \tau_2)\end{array}}{\Gamma \vdash \forall\ \overline{\alpha_1}^k.\ \tau_1 = \forall\ \overline{\alpha_2}^k.\ \tau_2} \quad (3.18)$$

Figure 3.5: *Valid and Equivalent Types.*

3.2.1 Type Scheme Instantiation

Before defining the typing judgement, it is necessary to define a number of functions on type schemes. In particular, we need a definition of an *instance* of a type scheme. Recall that a type-scheme can represent a polymorphic type (polytype). A type scheme σ' is called an instance, or *specialisation*, of a type scheme σ, if there exists a substitution $\mathcal{S}$ of types for type variables such that $\sigma' = \mathcal{S}\sigma$. We define a substitution $\mathcal{S}$ as a finite-map from type variables to types $\alpha \overset{map}{\mapsto} \tau$. If α are the free type variables of σ, then $\mathcal{S}\sigma$ is obtained by replacing each occurrence of α in σ with $\mathcal{S}(\alpha)$. Instantiation creates an instance of a polytype.

Instantiation of type schemes is performed by two functions *instance* and *subst* defined in Figure 3.6. The *instance* function is invoked with a type-scheme and a list of types to be substituted for the free type variables. If the type scheme is polymorphic, a substitution environment $\mathcal{S}$ is created, and the *subst* function is invoked to perform the substitutions. The *subst* function is defined recursively. There is a separate rule for each of the $\mathcal{M}\Lambda$ types. An actual substitution is only performed in the final rule for type variables α which occur in the domain of $\mathcal{S}$.. For later convenience, we also define functions *mono* and *tvs*. The *mono* function simply returns the type associated with a type scheme, i.e. excluding the type variables, while the *tvs* function returns the set of the quantified type variables for a type scheme, i.e. excluding the type.

$$\begin{array}{lcl} instance(\tau_1,\ \overline{\tau_2}^{k}) & = & \tau_1 \\ instance(\forall\,\overline{\alpha}^{k}.\ \tau_1,\ \overline{\tau_2}^{k}) & = & subst(\{\alpha^1\ \mapsto\ \tau_2^1,\ \ldots,\ \alpha^k\ \mapsto\ \tau_2^k\},\ \tau_1) \end{array} \tag{3.19}$$

$$\begin{array}{lcl} subst(\mathcal{S},\ tn) & = & tn \\ subst(\mathcal{S},\ tn(\overline{\tau}^{k})) & = & tn(subst(\mathcal{S},\ \overline{\tau}^{k})) \\ subst(\mathcal{S},\ \overline{\tau}^{k}) & = & (subst(\mathcal{S},\ \tau^1),\ \ldots,\ subst(\mathcal{S},\ \tau^k)) \\ subst(\mathcal{S},\ \tau_1 \rightarrow \tau_2) & = & subst(\mathcal{S},\ \tau_1) \rightarrow subst(\mathcal{S},\ \tau_2) \\ subst(\mathcal{S},\ \alpha) & = & \alpha \qquad (\alpha \notin \mathrm{Dom}\,\mathcal{S}) \\ subst(\mathcal{S},\ \alpha) & = & \mathcal{S}(\alpha) \qquad (\alpha \in \mathrm{Dom}\,\mathcal{S}) \end{array} \tag{3.20}$$

$$\begin{array}{lcl} mono(\tau) & = & \tau \\ mono(\forall\,\overline{\alpha}^{k}.\ \tau) & = & \tau \end{array} \tag{3.21}$$

$$\begin{array}{lcl} tvs(\tau) & = & \emptyset \\ tvs(\forall\,\overline{\alpha}^{k}.\ \tau) & = & \{\alpha^1,\ \ldots,\ \alpha^k\} \end{array} \tag{3.22}$$

Figure 3.6: *Type Scheme Instantiation.*

3.3 Typing Rules

The typing judgement $\Gamma \vdash P : \tau$ asserts that a program P is valid (i.e. type-able) and *has type* τ in Γ. This final judgement is the most complex and forms the essence of the type system. The top-level rule for this judgement is Rule: 3.23. Type checking of a program is performed in four stages, defined by the premise of this rule:

1. The initial environment Γ is extended with the datatype declarations $\Gamma[\overline{\overline{\overline{D}}}]$ yielding the environment Γ'. The set of datatypes are decomposed and entered into the environment according to Rule 3.24. There are separate cases for dealing with monomorphic and polymorphic datatypes. Each datatype declaration extends the environment with a single entry for the type name and a separate entry for each of the constructors. The functions *cname* (3.26) and *ctype* (3.27) provide the name and type of each constructor. The type of a nullary constructor is the parent datatype, while the type of a unary constructor is a function from the type of the constructor to the parent datatype. A constructor of a polymorphic datatype is universally quantified over the type variables of the parent datatype.

2. The environment is further extended with the exception declarations $\Gamma'[\overline{\overline{\overline{X}}}]$ according to Rule 3.25. This rule is similar to the rule for monomorphic datatype declarations. However, the exception declarations collectively form the constructors of a single monomorphic datatype t_exn.

3. The environment containing the datatypes and exceptions is checked for validity $\Gamma'' \vdash \diamond$.

4. Finally, the expression E is type-checked in the environment Γ''. The resulting type of the program is obtained from the type of this expression.

The rules in the remainder of this chapter determine the valid expressions and their types $\Gamma \vdash E : \tau$. There are many similarities between these rules. Therefore, rather than explaining each rule in turn, we will just highlight the main features. It is worth noting that each expression yields a monotype, or a monomorphic instance of a polytype. Polymorphism is hidden inside the **let** and **fix** expressions. This is called *let-bound* polymorphism, and is discussed in more detail in Chapter 4.

Programs

$$\frac{\Gamma' = \Gamma[\overline{\overline{D}}] \qquad \Gamma'' = \Gamma'[\overline{\overline{X}}] \qquad \Gamma'' \vdash \diamond \qquad \Gamma'' \vdash E : \tau}{\Gamma \vdash (\overline{\overline{D}}, \overline{\overline{X}}, E) : \tau} \tag{3.23}$$

$$\begin{array}{l}
\Gamma[\emptyset] = \Gamma \\
\Gamma[\{D^1, \ldots, D^k\}] = \Gamma[D^1] \cdots [D^k] \\
\Gamma[\textbf{datatype}\ \mathit{tn}\ \textbf{of}\ \{C^1, \ldots, C^k\}] = \\
\quad \Gamma[\mathit{tn} \mapsto (0, \{\mathit{cname}(C^1), \ldots, \mathit{cname}(C^k)\})] \\
\quad\ [\mathit{cname}(C^1) \mapsto \mathit{ctype}(C^1, \mathit{tn}), \ldots, \mathit{cname}(C^k) \mapsto \mathit{ctype}(C^k, \mathit{tn})] \\
\Gamma[\textbf{datatype}\ (\overline{\alpha}^k, \mathit{tn})\ \textbf{of}\ \{C^1, \ldots, C^l\}] = \\
\quad \Gamma[\mathit{tn} \mapsto (k, \{\mathit{cname}(C^1), \ldots, \mathit{cname}(C^l)\})] \\
\quad\ [\mathit{cname}(C^1) \mapsto \mathit{ctype}(C^1, (\overline{\alpha}^k, \mathit{tn})), \ldots, \\
\qquad \mathit{cname}(C^l) \mapsto \mathit{ctype}(C^l, (\overline{\alpha}^k, \mathit{tn}))]
\end{array} \tag{3.24}$$

$$\begin{array}{l}
\Gamma[\emptyset] = \Gamma \\
\Gamma[\{\textbf{exception}\ C^1, \ldots, \textbf{exception}\ C^k\}] = \\
\quad \Gamma[\mathrm{t\_exn} \mapsto (0, \{\mathit{cname}(C^1), \ldots, \mathit{cname}(C^k)\})] \\
\quad\ [\mathit{cname}(C^1) \mapsto \mathit{ctype}(C^1, \mathrm{t\_exn}), \ldots, \mathit{cname}(C^k) \mapsto \mathit{ctype}(C^k, \mathrm{t\_exn})]
\end{array} \tag{3.25}$$

$$\begin{array}{lcl}
\mathit{cname}(\mathit{con}) & = & \mathit{con} \\
\mathit{cname}((\mathit{con}, \tau)) & = & \mathit{con}
\end{array} \tag{3.26}$$

$$\begin{array}{lcl}
\mathit{ctype}(\mathit{con}, \mathit{tn}) & = & \mathit{tn} \\
\mathit{ctype}((\mathit{con}, \tau), \mathit{tn}) & = & \tau \to \mathit{tn} \\
\mathit{ctype}(\mathit{con}, (\overline{\alpha}^k, \mathit{tn})) & = & \forall\, \overline{\alpha}^k.\ \mathit{tn}(\overline{\alpha}^k) \\
\mathit{ctype}((\mathit{con}, \tau), (\overline{\alpha}^k, \mathit{tn})) & = & \forall\, \overline{\alpha}^k.\ \tau \to \mathit{tn}(\overline{\alpha}^k)
\end{array} \tag{3.27}$$

Values

$$\frac{}{\Gamma \vdash \textbf{scon}\ \mathit{scon} : \mathit{tn}_{\mathit{scon}}} \tag{3.28}$$

Comment: (Rule 3.28) The type of the special constant $\mathit{tn}_{\mathit{scon}}$ (e.g. t_int) is uniquely determined by the lexical structure of *scon*.

$$\frac{}{\Gamma \vdash \textbf{var}\ x : \Gamma(x)} \tag{3.29}$$

$$\frac{\Gamma \vdash \overline{\tau}^k}{\Gamma \vdash \textbf{var}\ (x, \overline{\tau}^k) : \mathit{instance}(\Gamma(x), \overline{\tau}^k)} \tag{3.30}$$

Comment: (Rule 3.30) All types which are explicitly specified in the syntax of language must be checked for validity. In this case, these types $\overline{\tau}^k$ are used to generate an instance of a polymorphic value.

$$\frac{\Gamma \vdash \tau_1 \rightarrow \tau_2 \qquad \Gamma[x \;\mapsto\; \tau_1] \vdash E : \tau_3 \qquad \Gamma \vdash \tau_2 = \tau_3}{\Gamma \vdash \mathbf{fn}\ (x, \tau_1 \rightarrow \tau_2) = E : \tau_1 \rightarrow \tau_2} \tag{3.31}$$

$$\frac{\Gamma \vdash \overline{\tau_1}^k \rightarrow \tau_2 \qquad \Gamma[x^1 \;\mapsto\; \tau_1^1,\ \ldots,\ x^k \;\mapsto\; \tau_1^k] \vdash E : \tau_3 \qquad \Gamma \vdash \tau_2 = \tau_3}{\Gamma \vdash \mathbf{fn}\ (\overline{x}^k, \overline{\tau_1}^k \rightarrow \tau_2) = E : \overline{\tau_1}^k \rightarrow \tau_2} \tag{3.32}$$

Comment: (Rule 3.32) This form of **fn** expression is an optimisation for functions of multiple arguments to prevent the unnecessary creation of closures.

Value Constructors

$$\frac{}{\Gamma \vdash \mathbf{con}\ \mathit{con} : \Gamma(\mathit{con})} \tag{3.33}$$

$$\frac{\Gamma(\mathit{con}) = \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash E : \tau_3 \qquad \Gamma \vdash \tau_1 = \tau_3}{\Gamma \vdash \mathbf{con}\ (\mathit{con}, E) : \tau_2} \tag{3.34}$$

Comment: (Rule 3.34) Constructing a unary datatype value is analogous to applying the constructor function $\tau_1 \rightarrow \tau_2$. The expression E must have the same type as the argument type τ_1, and the type of the entire expression is the result type τ_2.

$$\frac{\Gamma \vdash \overline{\tau}^k}{\Gamma \vdash \mathbf{con}\ (\mathit{con}, \overline{\tau}^k) : \mathit{instance}(\Gamma(\mathit{con}),\ \overline{\tau}^k)} \tag{3.35}$$

$$\frac{\Gamma \vdash \overline{\tau_1}^k \qquad \mathit{instance}(\Gamma(\mathit{con}),\ \overline{\tau_1}^k) = \tau_2 \rightarrow \tau_3 \qquad \Gamma \vdash E : \tau_4 \qquad \Gamma \vdash \tau_2 = \tau_4}{\Gamma \vdash \mathbf{con}\ (\mathit{con}, \overline{\tau_1}^k, E) : \tau_3} \tag{3.36}$$

$$\frac{\Gamma(\mathit{con}) = \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash E : \tau_3 \qquad \Gamma \vdash \tau_2 = \tau_3}{\Gamma \vdash \mathbf{decon}\ (\mathit{con}, E) : \tau_1} \tag{3.37}$$

Comment: (Rule 3.37) De-constructing a unary datatype value applies the constructor function $\tau_1 \rightarrow \tau_2$ in reverse. The expression E must have the same type as the result type τ_2, and the type of the entire expression is the argument type τ_1.

$$\frac{\Gamma \vdash \overline{\tau_1}^k \qquad \mathit{instance}(\Gamma(\mathit{con}),\ \overline{\tau_1}^k) = \tau_2 \rightarrow \tau_3 \qquad \Gamma \vdash E : \tau_4 \qquad \Gamma \vdash \tau_3 = \tau_4}{\Gamma \vdash \mathbf{decon}\ (\mathit{con}, \overline{\tau_1}^k, E) : \tau_2} \tag{3.38}$$

Structured Expressions

$$\frac{\Gamma \vdash E^1 : \tau^1 \quad \cdots \quad \Gamma \vdash E^k : \tau^k}{\Gamma \vdash \textbf{tuple}\ \overline{E}^k : \overline{\tau}^k} \tag{3.39}$$

$$\frac{\Gamma \vdash E : \overline{\tau}^k \qquad 0 < i \leq k}{\Gamma \vdash \textbf{select}\ (i,\, E) : \tau^i} \tag{3.40}$$

$$\frac{\Gamma \vdash \sigma \qquad \Gamma[tvs(\sigma)] \vdash E_1 : \tau_1 \qquad \Gamma \vdash mono(\sigma) = \tau_1 \qquad \Gamma[x \mapsto \sigma] \vdash E_2 : \tau_2}{\Gamma \vdash \textbf{let}\ (x,\, \sigma) = E_1\ \textbf{in}\ E_2 : \tau_2} \tag{3.41}$$

Comment: (Rule 3.41) Binding a polymorphic value to a variable requires some subtlety as expressions are not permitted to yield type-schemes. The expression E_1 must therefore return a type which matches the quantified type of σ. Since this type may contain free type variables and would therefore constitute an invalid type (according to our definition in Section 3.2) we must extend the environment with the type variables $tvs(\sigma)$ before checking E_1. The *mono* function is used to ensure that the resulting type is the same as the quantified type of σ. The type of the entire expression is obtained from the type of the expression E_2.

$$\frac{\begin{array}{l}\Gamma \vdash \sigma^1 \quad \cdots \quad \Gamma \vdash \sigma^k \\ \Gamma[tvs(\sigma^1)\ \cdots\ tvs(\sigma^k)] \vdash E_1 : \overline{\tau_1}^k \\ \Gamma \vdash mono(\sigma^1) = \tau_1^1 \quad \cdots \quad \Gamma \vdash mono(\sigma^k) = \tau_1^k \\ \Gamma[x^1 \mapsto \sigma^1,\, \ldots,\, x^k \mapsto \sigma^k] \vdash E_2 : \tau_2\end{array}}{\Gamma \vdash \textbf{let}\ \overline{(x,\, \sigma)}^k = E_1\ \textbf{in}\ E_2 : \tau_2} \tag{3.42}$$

$$\frac{\begin{array}{l}\Gamma \vdash E_1 : \tau_1 \qquad \Gamma \vdash \tau_1 = \tau_{c^1} = \cdots = \tau_{c^k} \\ \Gamma \vdash E_2^1 : \tau_2^1 \quad \cdots \quad \Gamma \vdash E_2^k : \tau_2^k \\ \Gamma \vdash E_3 : \tau_3 \qquad \Gamma \vdash \tau_3 = \tau_2^1 = \cdots = \tau_2^k\end{array}}{\Gamma \vdash \textbf{switch}\ E_1\ \textbf{case}\ (\{c^1 \mapsto E_2^1,\, \ldots,\, c^k \mapsto E_2^k\},\, E_3) : \tau_3} \tag{3.43}$$

Comment: (Rule 3.43) We denote the type of a constant c, which represents either a special constant or a constructor, by τ_c. The **switch** expression contains a finite map from constants to expressions. The constants in the domain of this finite-map must all have the same type, which must be equal to the type of the condition expression E_1. Similarly, the expressions in the range of the finite map must all have the same type, which must be equal to the type of the default expression E_3.

Function Expressions

$$\frac{\begin{array}{l}\Gamma \vdash \sigma^1 \quad \cdots \quad \Gamma \vdash \sigma^k \\ \Gamma' = \Gamma[\mathit{tvs}(\sigma^1) \ \cdots \ \mathit{tvs}(\sigma^k)][x^1 \ \mapsto \ \mathit{mono}(\sigma^1), \ \ldots, \ x^k \ \mapsto \ \mathit{mono}(\sigma^k)] \\ \Gamma' \vdash E_1^1 : \tau_1^1 \quad \cdots \quad \Gamma' \vdash E_1^k : \tau_1^k \\ \Gamma \vdash \mathit{mono}(\sigma^1) = \tau_1^1 \quad \cdots \quad \Gamma \vdash \mathit{mono}(\sigma^k) = \tau_1^k \\ \Gamma[x^1 \ \mapsto \ \sigma^1, \ \ldots, \ x^k \ \mapsto \ \sigma^k] \vdash E_2 : \tau_2\end{array}}{\Gamma \vdash \textbf{fix} \ \overline{(x, \sigma) = E_1}^{k} \ \textbf{in} \ E_2 : \tau_2} \tag{3.44}$$

Comment: (Rule 3.44) The fixed-point expression uses the same technique as the **let** expression (3.41) to enable the binding of polymorphic function values to variables. However, as any function may refer to any other, we check the function expressions in an environment Γ' which already contains all of the function bindings.

$$\frac{\Gamma \vdash E_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash E_2 : \tau_3 \qquad \Gamma \vdash \tau_1 = \tau_3}{\Gamma \vdash \textbf{app} \ (E_1, E_2) : \tau_2} \tag{3.45}$$

$$\frac{\begin{array}{l}\Gamma \vdash E_1 : \overline{\tau_1}^{k} \to \tau_2 \qquad \Gamma \vdash E_2^1 : \tau_3^1 \quad \cdots \quad \Gamma \vdash E_2^k : \tau_3^k \\ \Gamma \vdash \tau_1^1 = \tau_3^1 \quad \cdots \quad \Gamma \vdash \tau_1^k = \tau_3^k\end{array}}{\Gamma \vdash \textbf{app} \ (E_1, \overline{E_2}^{k}) : \tau_2} \tag{3.46}$$

References and Exceptions

$$\frac{\begin{array}{l}\Gamma \vdash E_1 : \tau_1 \qquad \Gamma \vdash E_2 : \tau_2 \\ \mathit{instance}(\Gamma(\text{c\_ref}), \ (\tau_2)) = \tau_3 \to \tau_4 \qquad \Gamma \vdash \tau_1 = \tau_4\end{array}}{\Gamma \vdash \textbf{assign} \ (E_1, E_2) : \text{t\_unit}} \tag{3.47}$$

$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash E : \text{t\_exn}}{\Gamma \vdash \textbf{raise} \ (E, \tau) : \tau} \tag{3.48}$$

Comment: (Rule 3.48) The supplied type τ provides the type of the **raise** expression, as the type t_exn here would, in general, cause a type error. This is simply a convenience to simplify the typing rules.

$$\frac{\Gamma \vdash E_1 : \tau_1 \qquad \Gamma \vdash E_2 : \text{t\_exn} \to \tau_3 \qquad \Gamma \vdash \tau_1 = \tau_3}{\Gamma \vdash \textbf{handle} \ E_1 \ \textbf{with} \ E_2 : \tau_1} \tag{3.49}$$

Threads

$$\frac{\Gamma \vdash E : \text{t\_unit} \rightarrow \text{t\_unit}}{\Gamma \vdash \textbf{fork}\ E : \text{t\_unit}} \tag{3.50}$$

$$\frac{\Gamma \vdash E_1 : \text{t\_int} \qquad \Gamma \vdash E_2 : \text{t\_unit} \rightarrow \text{t\_unit}}{\Gamma \vdash \textbf{rfork}\ (E_1, E_2) : \text{t\_unit}} \tag{3.51}$$

Comment: (Rule 3.51) The expression E_1 evaluates to an integer which determines the remote processor on which E_2 should be executed. For convenience we assume that any integer can denote a processor. We can easily map arbitrary integers onto processor numbers using $i\ mod\ n$, where i is the integer value and n is the maximum number of processors.

$$\frac{\begin{array}{l}\Gamma \vdash E_1 : \tau_1 \qquad \Gamma \vdash E_2 : \tau_2 \\ \mathit{instance}(\Gamma(\text{c\_chan}),\ (\tau_2)) = \tau_3 \rightarrow \tau_4 \qquad \Gamma \vdash \tau_1 = \tau_4\end{array}}{\Gamma \vdash \textbf{send}\ (E_1, E_2) : \text{t\_unit}} \tag{3.52}$$

Comment: (Rule 3.51) Compare with Rule 3.47; sending a value across a polymorphic channel is essentially the same as assigning a value to a polymorphic reference.

$$\frac{\Gamma \vdash E : \text{t\_chan}(\tau)}{\Gamma \vdash \textbf{receive}\ E : \tau} \tag{3.53}$$

3.4 Type Checking

In Chapter 1 we stated that a type system is specified separately from the algorithm used to perform the type checking. In general, it is useful to decouple the type system from the type checking-algorithm. The type system is part of the language definition, while the type-checking algorithm is part of the compiler. Moreover, it is often possible to define different type checking algorithms for the same type system. However, it is also possible to define type systems for which no type checking algorithm can be constructed.

In this chapter we have defined the type system for $\mathcal{M}\Lambda$. This type system is relatively straightforward as it is completely deterministic. Consequently, $\mathcal{M}\Lambda$ can be type-checked with a simple pattern-matching depth-first algorithm, though we do not give the details here. In Figure 3.7 we illustrate the type checking process on the identity function.

$$\underbrace{\alpha \in \mathrm{Dom}\,\Gamma'''}_{(3.11)} \quad \underbrace{\alpha \in \mathrm{Dom}\,\Gamma'''}_{(3.11)} \quad \underbrace{\alpha \in \mathrm{Dom}\,\Gamma'''}_{(3.11)} \quad \underbrace{\alpha \in \mathrm{Dom}\,\Gamma'''}_{(3.11)}$$

$$\underbrace{\Gamma''' \vdash \alpha \qquad \Gamma''' \vdash \alpha}_{(3.10)} \qquad \underbrace{\Gamma''' \vdash \alpha \qquad \Gamma''' \vdash \alpha}_{(3.10)} \qquad \underbrace{\Gamma''' \vdash \boxed{\mathbf{var}\ x} : \Gamma'''(x)}_{(3.29)} \qquad \underbrace{\alpha \in \mathrm{Dom}\,\Gamma'''}_{(3.17)} \qquad \underbrace{\vdots}$$

$$\underbrace{\Gamma''[\alpha] \vdash \alpha \to \alpha}_{(3.12)} \qquad \underbrace{\Gamma''[\alpha] \vdash \alpha \to \alpha \qquad \Gamma''[\alpha][x \mapsto \alpha] \vdash \boxed{\mathbf{var}\ x} : \alpha \qquad \Gamma''[\alpha] \vdash \alpha = \alpha}_{(3.31)} \qquad \underbrace{\Gamma''' \vdash \boxed{\ldots} : \tau}$$

$$\underbrace{\Gamma[\emptyset] = \Gamma}_{(3.24)} \quad \underbrace{\Gamma'[\emptyset] = \Gamma'}_{(3.25)} \quad \underbrace{\Gamma'' \vdash \forall(\alpha).\ \alpha \to \alpha \qquad \Gamma''[\alpha] \vdash \boxed{\mathbf{fn}\ (x,\ \alpha \to \alpha)\ =\ \mathbf{var}\ x} : \alpha \to \alpha \qquad \Gamma'' \vdash \alpha \to \alpha = \alpha \to \alpha \qquad \Gamma''[x \mapsto \forall(\alpha).\ \alpha \to \alpha] \vdash \boxed{\ldots} : \tau}_{(3.41)}$$

$$\underbrace{\Gamma' = \Gamma[\emptyset] \qquad \Gamma'' = \Gamma'[\emptyset] \qquad \Gamma'' \vdash \diamond\ \text{(follows from } \Gamma \vdash \diamond) \qquad \Gamma'' \vdash \boxed{\mathbf{let}\ (\text{ident},\ \forall(\alpha).\ \alpha \to \alpha)\ =\ (\mathbf{fn}\ (x,\ \alpha \to \alpha)\ =\ \mathbf{var}\ x)\ \mathbf{in}\ \ldots} : \tau}_{(3.23)}$$

$$\Gamma \vdash (\boxed{\emptyset,\ \emptyset,\ \mathbf{let}\ (\text{ident},\ \forall(\alpha).\ \alpha \to \alpha)\ =\ (\mathbf{fn}\ (x,\ \alpha \to \alpha)\ =\ \mathbf{var}\ x)\ \mathbf{in}\ \ldots}) : \tau$$

Figure 3.7: *Type Checking Example.*

3.5 System F

The representation of polymorphism in the $\mathcal{M}\Lambda$ intermediate language is particularly important. There are many different styles of polymorphism, and we must be careful to ensure that the polymorphism in $\mathcal{M}\Lambda$ is a suitable match for that found in Standard ML (and hence Dynamic ML). Sound type systems for polymorphic languages are particularly difficult to construct. Therefore, the type systems of a variety of typed λ-calculi have been the object of much theoretical study over many years. A hierarchy of typed λ-calculi has been developed, called System F [PDM89]. The first-order simply-typed (non-polymorphic) λ-calculus is classified as F_1. The second-order polymorphic λ-calculus of Girard-Reynolds [Rey74] is classified as F_2. This is an extension of F_1 with *type-parameters*, i.e. functions from types to expressions. There are further higher-order typed λ-calculi: F_3, F_4, etc. These calculi are extensions of F_2 that allow functions from types to types. However, in order to keep the types of these functions under control, a new layer of types for the type themselves are introduced called *kinds*.

The polymorphism which is found in Dynamic ML is defined by F_2. Consequently, we have also chosen F_2 polymorphism as the basis of $\mathcal{M}\Lambda$. Nonetheless, this choice is not as obvious as it seems. The entire System F hierarchy can be represented by the ω-order polymorphic λ-calculus defined by F_ω. The language F_ω differs from F_3, F_4, etc. only in that the set of kinds is larger. There is a certain intuitive appeal for using F_ω as the basis for our intermediate language. Indeed, the FLINT/ML intermediate language [Sha97] used in the Standard ML of New Jersey is based on F_ω. However, after careful consideration we decided not to take this approach. FLINT/ML is intended as a general purpose intermediate representation for a range of programming languages. Thus, the generality provided by F_ω may prove useful. On the other hand, $\mathcal{M}\Lambda$ is intended as a simple language on which we will define a number of operations in a clean manner while avoiding a great deal of unnecessary complexity associated with Dynamic ML. Consequently, if we were to base our intermediate language on a kind of polymorphism that is more general than that found in Dynamic ML we would be increasing, rather than decreasing, the complexity of the task. Thus, F_2 appears to be the correct choice for our purposes. In the remainder of this chapter we will present the syntax, semantics, and type system for a call-by-value variant of F_2 and explain its relation to $\mathcal{M}\Lambda$.

$\sigma \in \text{Type}$	$::=$	α	(type variable)
	$\mid$	$\sigma_1 \to \sigma_2$	(function type)
	$\mid$	$\forall\alpha.\ \sigma$	(polymorphic type)
$e \in \text{Expression}$	$::=$	x	(lambda variable)
	$\mid$	$\lambda x : \sigma.\ e$	(function abstraction)
	$\mid$	$e_1\ e_2$	(function application)
	$\mid$	$\Lambda\alpha.\ e$	(type abstraction)
	$\mid$	$e[\sigma]$	(type application)

Figure 3.8: *Syntax of F$_2$.*

The syntax of F$_2$ is defined by the grammar presented in Figure 3.8. The calculus contains two syntactic classes: types σ, and expressions e. In $\mathcal{M}\Lambda$, as in F$_2$, the types are separated from the expressions. We will refer to the types collectively as the type language, and the expressions collectively as the expression language. Although the $\mathcal{M}\Lambda$ and F$_2$ type languages are broadly similar, there are some important differences discussed below.

At first glance, the $\mathcal{M}\Lambda$ expression language appears completely different from F$_2$. However, this is primarily due to the addition of user-defined datatypes and exceptions into the language, and the adoption of ML-style keywords and other syntactic-sugaring. Figure 3.9 illustrates the basic correspondence between the F$_2$ expressions and their $\mathcal{M}\Lambda$ counterparts. There are other matches between the languages, not illustrated in this table. For example, a polymorphic datatype declaration is effectively a type abstraction, and a constructor for a polymorphic datatype is effectively a type application. It would be possible to encode the entire $\mathcal{M}\Lambda$ language in F$_2$. However, the new expression forms, such as multiple-argument functions, make $\mathcal{M}\Lambda$ more efficient as a typed intermediate language of a compiler.

F$_2$ Expression e	$\mathcal{M}\Lambda$ Expression E
x	**var** x
$\lambda x : \sigma.\ e$	**fn** $(x,\ \tau_1 \to \tau_2)\ =\ E$
$e_1\ e_2$	**app** $(E_1,\ E_2)$
$\Lambda\alpha.\ e$	**let** $(x,\ \sigma)\ =\ E_1$ **in** E_2
$e[\sigma]$	**var** $(x,\ \overline{\tau}^k)$

Figure 3.9: *Relationship between F$_2$ and $\mathcal{M}\Lambda$ expressions.*

In F_2, polytypes and monotypes are not distinguished. This complicates the semantics and makes the calculus ‘circular’ or *impredicative.* In an impredicative system, the definition of an object can refer to the collection to which the object belongs. For example, the polymorphic identity $\Lambda\alpha.\ (\lambda x : \alpha.\ x)$ can take its own type $\alpha \to (\alpha \to \alpha)$ and then itself as arguments. By contrast, a predicative system is structured into levels so that each object is defined in terms of components that belongs to a level smaller than the one to which the defined object itself belongs. In $\mathcal{M}\Lambda$ we take the standard approach of splitting the type hierarchy into two levels: a type level τ which expresses the monomorphic types, and a type scheme level σ which expresses the polymorphic types. This distinction leaves us with a predicative type language. In our predicative type language we can no longer embed universally quantified types. We therefore alter the universally quantified type to take a sequence of type variables, e.g. $(\forall\alpha.\ (\forall\alpha'.\ (\forall\alpha''.\ \sigma)))$ is represented in $\mathcal{M}\Lambda$ as $\forall(\alpha,\ \alpha',\ \alpha'').\ \tau$. Since we do not allow empty sequences, we provide a separate monomorphic type in the definition of type schemes to represent an instance.

At the monomorphic type level, we have enriched the language with user defined type names *tn* and $tn(\overline{\tau}^k)$. Recall that the type names *tn* include the basic types, as required by the special constants; value constructor types; reference types; and exception types. The second form $tn(\overline{\tau}^k)$ is used for representing polymorphic datatypes e.g. t_list(α). For convenience, we also include a tuple type $\overline{\tau}^k$ which can be used to type records and functions with more than one argument.

$$\frac{}{\emptyset \vdash \diamond} \quad (3.54) \qquad \frac{\alpha \in \mathrm{Dom}\,\Gamma}{\Gamma \vdash \alpha} \quad (3.57)$$

$$\frac{\Gamma \vdash \sigma \quad x \notin \mathrm{Dom}\,\Gamma}{\Gamma[x\ \mapsto\ \sigma] \vdash \diamond} \quad (3.55) \qquad \frac{\Gamma \vdash \sigma_1 \quad \Gamma \vdash \sigma_2}{\Gamma \vdash \sigma_1 \to \sigma_2} \quad (3.58)$$

$$\frac{\Gamma \vdash \diamond \quad \alpha \notin \mathrm{Dom}\,\Gamma}{\Gamma[\alpha] \vdash \diamond} \quad (3.56) \qquad \frac{\Gamma[\alpha] \vdash \sigma}{\Gamma \vdash \forall\alpha.\ \sigma} \quad (3.59)$$

Figure 3.10: *Valid Environments* $\Gamma \vdash \diamond$ *and Types* $\Gamma \vdash \sigma$*.*

We will now look briefly at the type system for F$_2$ upon which our $\mathcal{M}\Lambda$ type system, presented in this chapter, is based. The typing judgements for valid environments $\Gamma \vdash \diamond$ and valid types $\Gamma \vdash \sigma$ are defined in Figure 3.10. Note that the typing environment Γ in F$_2$ contains a set of type variables $\overline{\overline{\alpha}}$ and a mapping between lambda variables and their types $x \mapsto \sigma$. It is immediately apparent that the equivalent $\mathcal{M}\Lambda$ typing rules in Figures 3.4 and 3.5 are a direct superset of the rules in F$_2$. The typing judgement for the F$_2$ expressions $\Gamma \vdash e : \sigma$ is defined in Figure 3.11. Again, it is clear that the rules for the equivalent forms in $\mathcal{M}\Lambda$ are very similar to those found in F$_2$. However, in $\mathcal{M}\Lambda$ as in Dynamic ML, polymorphism is let-bound and all of the expressions are typed to monomorphic or instantiated polymorphic types.

The connection between the $\mathcal{M}\Lambda$ type system and the F$_2$ type system upon which it is based is clearly apparent. We therefore argue that the $\mathcal{M}\Lambda$ type system is likely to be sound, as the type systems of F$_2$ is known to be sound, though we do not give a proof here. In the following chapter we define the translation from Dynamic ML to $\mathcal{M}\Lambda$ and thereby reinforce the connection between Dynamic ML and the underlying type theory.

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad (3.60)$$

$$\frac{\Gamma[x \mapsto \sigma_1] \vdash e : \sigma_2}{\Gamma \vdash \lambda x : \sigma_1.\ e : \sigma_1 \to \sigma_2} \qquad (3.61)$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \sigma_1 \to \sigma_2 \\ \Gamma \vdash e_2 : \sigma_1\end{array}}{\Gamma \vdash e_1\ e_2 : \sigma_2} \qquad (3.62)$$

$$\frac{\Gamma[\alpha] \vdash e : \sigma}{\Gamma \vdash \Lambda\alpha.\ e : \forall\alpha.\ \sigma} \qquad (3.63)$$

$$\frac{\Gamma \vdash \sigma_1 \quad \Gamma \vdash e : \forall\alpha.\ \sigma_2}{\Gamma \vdash e[\sigma_1] : \mathit{subst}(\{\alpha \mapsto \sigma_1\},\ \sigma_2)} \qquad (3.64)$$

Figure 3.11: *Well-Formed Expressions* $\Gamma \vdash e : \sigma$

Chapter 4

Translating Dynamic ML to $\mathcal{M}\Lambda$

An outline of the translation from Dynamic ML into the $\mathcal{M}\Lambda$ intermediate language is presented in this chapter. In previous chapters we have exhibited a number of program examples which illustrate the translation process. We will now reinforce this material by taking a formal approach to the translation. This is intended to convince the reader that the $\mathcal{M}\Lambda$ language is indeed a suitable candidate for the compilation of Dynamic ML. In later chapters we will use the simpler $\mathcal{M}\Lambda$ language to express a number of definitions which would be cumbersome in terms of the full language.

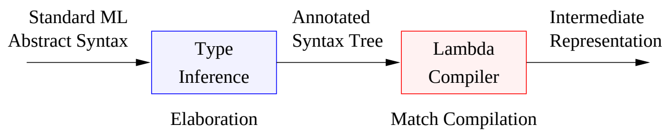


Figure 4.1: *Two-Step Translation.*

The translation is defined by a set of inference rules, in a similar style to the typing rules of Chapter 3. The Dynamic ML language contains pattern matching, and type information is inferred (i.e. implicit). In order to effect the translation into $\mathcal{M}\Lambda$ it is necessary to perform an *elaboration* step to generate explicit type information, and a *match-compilation* step to convert pattern matching into simpler switch statements (Figure 4.1). Both of these operations are well-understood, and a variety of different techniques exist. In this chapter we adapt these techniques to the translation into $\mathcal{M}\Lambda$. We have presented the translation process in a very straightforward manner. Thus, the resulting $\mathcal{M}\Lambda$ programs are somewhat suboptimal. In order to generate efficient programs, the usual lambda language optimisations, e.g. lambda lifting, closure elimination, constant folding [App92], should be applied to the resulting programs, though we do not give the details here.

4.1 Dynamic ML Syntax

The abstract syntax of the Dynamic ML core language is defined in Figure 4.2. The base syntactic categories of the language are type variables *tyvar*, type constructors *tycon*, value identifiers *vid*, record labels *lab*, and special constants *scon*. Optional elements are enclosed in angled brackets $\langle\ \rangle$. Sequences are represented as dotted $\cdots$ lists, they differ from the earlier $\mathcal{M}\Lambda$ definition in that empty sequences are permitted.

ty	::=	*tyvar*	Type Variable
	|	`{`lab_1 `:` ty_1 `,` $\cdots$ `,` lab_n `:` ty_n`}`	Record Type
	|	`(`ty_1 `,` $\cdots$ `,` ty_n`)` *tycon*	Type Construction
	|	ty_1 `->` ty_2	Function Type
program	::=	*dec*	
dec	::=	`val` `(`$tyvar_1$ `,` $\cdots$ `,` $tyvar_n$`)` *pat* `=` *expr*	Value
	|	`val` `(`$tyvar_1$ `,` $\cdots$ `,` $tyvar_n$`)` `rec` *pat* `=` *expr*	Recursive Value
	|	dec_1 $\langle$`;`$\rangle$ dec_2	Sequencing
	|	`local` dec_1 `in` dec_2 `end`	Local
	|	`exception` *vid* $\langle$`of` *ty*$\rangle$	Exception
	|	`datatype` *datbind*	Datatype
datbind	::=	`(`$tyvar_1$ `,` $\cdots$ `,` $tyvar_n$`)` *tycon* `=` *conbind* $\langle$`and` *datbind*$\rangle$	
conbind	::=	*vid* $\langle$`of` *ty*$\rangle$ $\langle$`|` *conbind*$\rangle$	
expr	::=	*scon*	Special Constant
	|	*vid*	Value Identifier
	|	`{`lab_1 `=` $expr_1$ `,` $\cdots$ `,` lab_n `=` $expr_n$`}`	Record
	|	`let` *dec* `in` *expr* `end`	Binding
	|	$expr_1$ $expr_2$	Application
	|	*expr* `:` *ty*	Type Annotation
	|	*expr* `handle` *match*	Handle Exception
	|	`raise` *expr*	Raise Exception
	|	`fn` *match*	Function
match	::=	*pat* `=>` *expr* $\langle$`|` *match*$\rangle$	
pat	::=	`_`	Wildcard
	|	*scon*	Constant
	|	*vid*	Identifier
	|	`{`lab_1 `=` pat_1 `,` $\cdots$ `,` lab_n `=` pat_n`}`	Record
	|	*pat* `:` *ty*	Typed
	|	*vid* *pat*	Constructed
	|	*vid* $\langle$`:` *ty*$\rangle$ `as` *pat*	Layered

Figure 4.2: *Dynamic ML Abstract Syntax.*

We only define the Dynamic ML core language in this chapter. The Dynamic ML modules language is presented in Chapter 7. We have made a number of simplifications and minor changes to the syntax of the core language presented in the *Definition of Standard ML* [MTHM97], which are detailed below:

1. A *program* is defined as a single declaration *dec*. Complex programs can be built using the sequencing `;` operation.

2. All of the *derived forms* described in Appendix A of [MTHM97] have been removed from the language, e.g. tuples are represented as records, `fun` is represented by `val rec`, etc.

3. For clarity, the syntactic categories have been slightly rearranged. For example, we do not distinguish the atomic forms of patterns and expressions.

4. The `abstype` declaration has been removed as it is essentially deprecated in the 1997 revision of Standard ML. Opaque signature matching at the module level can now be used in place of `abstype`. In Chapter 7 we will discuss how module-level constructs can be synthesised without requiring any extra code to be generated.

5. We require that all identifiers *vid* (including type variables α) are unique, i.e. the same identifier must not be bound more than once. This can be accomplished by a simple renaming process. We also require that no record expression, pattern or type contains duplicate field labels.

6. Type abbreviations and datatype/exception replication have been removed as they can largely be implemented by a straightforward syntactic substitution. Abbreviated record patterns (using `...` notation) are not included as they significantly complicate the translation process by requiring incomplete tuple types.

7. Any type variable appearing in a *conbind* must also appear in the parent *datbind*.

8. In a `val rec` *pat* `=` *expr* declaration, the pattern *pat* must be of the form:

   $$\texttt{\{}lab_1 \texttt{ = } vid_1\texttt{,}\ \cdots\texttt{,}\ lab_n \texttt{ = } vid_n\texttt{\}}$$

   and the expression *expr* must be of the form:

   $$\texttt{\{}lab_1 \texttt{ = fn } match_1\texttt{,}\ \cdots\texttt{,}\ lab_n \texttt{ = fn } match_n\texttt{\}}$$

4.2 Two-Step Translation

As stated earlier, the translation from Dynamic ML to $\mathcal{M}\Lambda$ will be performed in two steps; a elaboration step, and a match-compilation step. These steps could in principle be combined into a single operation, since both perform a straightforward depth-first traversal of the abstract-syntax tree. However, the presentation is significantly clearer if we treat these operations independently. The translation from Dynamic ML abstract-syntax to $\mathcal{M}\Lambda$ code, written $\mathit{program} \triangleright P$, is captured by Rule 4.1. The premise of this rule corresponds to the elaboration operation (:) and match-compilation operation ($\rightsquigarrow$). The remainder of this chapter is devoted to providing a definition of these two rules.

$$\frac{\Gamma \vdash \mathit{program} : (\Gamma',\ \mathcal{S}) \qquad \Gamma',\ \mathcal{S} \vdash \mathit{program}' \rightsquigarrow P \\ \mid \mathit{program}' \mid = \mathit{program}}{\Gamma \vdash \mathit{program} \triangleright P} \tag{4.1}$$

Comment: (Rule 4.1) During type inference, the abstract syntax tree is annotated with all of the intermediate type information. Hence, the input to the translation step is the annotated tree, which we denote $\mathit{program}'$. For clarity, we have omitted the definition of this annotation process from the rules, though the annotation process is illustrated in the example shown towards the end of this chapter. The side-condition $\mid \mathit{program}' \mid = \mathit{program}$ means that the annotated $\mathit{program}'$ is identical to the initial $\mathit{program}$ under the removal of all annotations, i.e. annotation does not change the meaning or form of the program.

4.3 Type Inference

We will initially turn our attention to the translation of Dynamic ML type information into $\mathcal{M}\Lambda$ type information. We have previously stated that the type information in Dynamic ML is implicit and is obtained by elaboration. Nonetheless, this does not tell the whole story. Dynamic ML contains both explicit and implicit type information. Explicit type information is included directly in the syntax of the Dynamic ML program, in the form of type annotations, and therefore does not need to be inferred. This explicit type information is represented by the syntactic category ty in Figure 4.2. Therefore, before we tackle the more difficult elaboration problem, we will define the translation of the explicit types ty into $\mathcal{M}\Lambda$ types τ.

The translation of the explicit types is captured by the judgement $\Gamma \vdash ty : \tau$. The environment Γ is the static typing environment from Chapter 3. The type *ty* is a Dynamic ML type from Figure 4.2, and type type τ is the equivalent $\mathcal{M}\Lambda$ type. The rules for this judgement are given below. We assume that Γ contains the initial static environment from Section 3.1.1. Note that Dynamic ML type variables *tyvar* and type constructors *tycon* are directly equivalent to $\mathcal{M}\Lambda$ type variables α and type names *tn* respectively. We will express the conversion using the notation: $\mathit{tyvar} \leadsto \alpha$ and $\mathit{tycon} \leadsto \mathit{tn}$.

$$\frac{\mathit{tyvar} \leadsto \alpha}{\Gamma \vdash \mathit{tyvar} : \alpha} \tag{4.2}$$

$$\frac{}{\Gamma \vdash \texttt{\{\}} : \text{t\_unit}} \tag{4.3}$$

Comment: (Rule 4.3) The unit type is represented by the empty record type in Dynamic ML. We translate this into the $\mathcal{M}\Lambda$ type t_unit.

$$\frac{\begin{array}{l}\Gamma \vdash \mathit{ty}_1 : \tau_1 \quad \cdots \quad \Gamma \vdash \mathit{ty}_n : \tau_n \\ \delta \text{ a permutation of } 1 \ldots n, \text{ where } \mathit{lab}_{\delta(1)} < \cdots < \mathit{lab}_{\delta(n)}\end{array}}{\Gamma \vdash \texttt{\{}\mathit{lab}_1 \texttt{ : } \mathit{ty}_1\texttt{,} \cdots \texttt{,} \; \mathit{lab}_n \texttt{ : } \mathit{ty}_n\texttt{\}} : (\tau_{\delta(1)}, \ldots, \tau_{\delta(n)})} \tag{4.4}$$

Comment: (Rule 4.4) Records are represented by ordered tuples in $\mathcal{M}\Lambda$. We sort the labels lexicographically (using $<$) with a permutation function δ.

$$\frac{\mathit{tycon} \leadsto \mathit{tn} \qquad \Gamma(\mathit{tn}) = (0, \overline{\overline{\mathit{con}}})}{\Gamma \vdash \texttt{()} \; \mathit{tycon} : \mathit{tn}} \tag{4.5}$$

Comment: (Rule 4.5) If no types are supplied, we check that the arity of the type constructor is 0. The elaboration of monomorphic datatypes is defined in Rule 4.25.

$$\frac{\Gamma \vdash \mathit{ty}_1 : \tau_1 \quad \cdots \quad \Gamma \vdash \mathit{ty}_n : \tau_n \qquad \mathit{tycon} \leadsto \mathit{tn} \qquad \Gamma(\mathit{tn}) = (n, \overline{\overline{\mathit{con}}})}{\Gamma \vdash \texttt{(}\mathit{ty}_1\texttt{,} \cdots \texttt{,} \; \mathit{ty}_n\texttt{)} \; \mathit{tycon} : \mathit{tn}(\tau_1, \ldots, \tau_n)} \tag{4.6}$$

Comment: (Rule 4.5) When constructing an instance of a polymorphic datatype, the number of types n must match the arity of the datatype. The elaboration of polymorphic datatypes is defined in Rule 4.26.

$$\frac{\Gamma \vdash \mathit{ty}_1 : \tau_1 \qquad \Gamma \vdash \mathit{ty}_2 : \tau_2}{\Gamma \vdash \mathit{ty}_1 \texttt{ -> } \mathit{ty}_2 : \tau_1 \to \tau_2} \tag{4.7}$$

4.3.1 Algorithm $\mathcal{W}$

We will now turn our attention to the implicit types of Dynamic ML. We must convert these implicit types into explicit $\mathcal{M}\Lambda$ types through a process of elaboration. The type system for Standard ML is presented in the Definition [MTHM97]. However, as discussed in Chapter 3, the type system alone is usually insufficient to determine the actual type information, due to non-determinism present in the rules. For the Dynamic ML type system, we need to use a variant of Milner's algorithm $\mathcal{W}$ [Mil78] to obtain the actual types of the expressions. This algorithm finds the *principal* (most polymorphic) type for an expression, if one exists. We will adapt this algorithm to elaborate the implicit Dynamic ML types into explicit $\mathcal{M}\Lambda$ types.

The definition of algorithm $\mathcal{W}$, which we will shortly present, is centred on the unification algorithm of Robinson [Rob65]. The unification algorithm takes a pair of types $(\tau_1,\ \tau_2)$ and returns the most general unifier; a substitution $\mathcal{S}$ from type variables to types (Section 3.2.1), such that $\mathit{subst}(\mathcal{S},\ \tau_1) = \mathit{subst}(\mathcal{S},\ \tau_2)$. A particular unification may also fail to have a unifier, in which case the inference algorithm will fail. Failure of unification is typically used in an ML compiler to report a type error to the programmer. The unification algorithm for the $\mathcal{M}\Lambda$ types is presented in the *unify* function defined below.

$$\frac{}{\vdash unify(tn,\ tn) = \emptyset} \tag{4.8}$$

$$\frac{\vdash unify(\overline{\tau_1}^k,\ \overline{\tau_2}^k) = \mathcal{S}}{\vdash unify(tn(\overline{\tau_1}^k),\ tn(\overline{\tau_2}^k)) = \mathcal{S}} \tag{4.9}$$

Comment: (Rules 4.8 and 4.9) Unifying a type name *tn* with itself (Rule 4.8) succeeds, but results in an empty substitution as there are no type variables involved. For a constructed type (Rule 4.9), we must also check that the tuple of types unify correctly.

$$\frac{\begin{array}{l}\vdash unify(\tau_1^1,\ \tau_2^1) = \mathcal{S}_1 \\ \vdash unify(subst(\mathcal{S}_1,\ \tau_1^2),\ subst(\mathcal{S}_1,\ \tau_2^2)) = \mathcal{S}_2 \quad \cdots \\ \quad \vdash unify(subst(\mathcal{S}_{k-1},\ \tau_1^k),\ subst(\mathcal{S}_{k-1},\ \tau_2^k)) = \mathcal{S}_k\end{array}}{\vdash unify(\overline{\tau_1}^k,\ \overline{\tau_2}^k) = \mathcal{S}_k \cdots \mathcal{S}_1} \tag{4.10}$$

Comment: (Rule 4.10) We perform unification of tuples in left-to-right order. Each substitution is applied to the remaining types before they are unified. The result is the composition of all the substitutions.

$$\frac{\vdash unify(\tau_1,\ \tau_1') = \mathcal{S}_1 \qquad \vdash unify(subst(\mathcal{S}_1,\ \tau_2),\ subst(\mathcal{S}_1,\ \tau_2')) = \mathcal{S}_2}{\vdash unify(\tau_1 \rightarrow \tau_2,\ \tau_1' \rightarrow \tau_2') = \mathcal{S}_2\mathcal{S}_1} \tag{4.11}$$

$$\frac{}{\vdash unify(\alpha,\ \alpha) = \emptyset} \tag{4.12}$$

Comment: (Rule 4.12) Unifying a type variable with itself succeeds and results in an empty substitution.

$$\frac{\alpha \notin ftvs(\tau)}{\vdash unify(\alpha,\ \tau) = \{\alpha\ \mapsto\ \tau\}} \tag{4.13}$$

Comment: (Rule 4.13) The condition $\alpha \notin ftvs(\tau)$ is called the *occurs check* and is used to eliminate cyclic types. The $ftvs(\tau)$ function is defined in 4.15.

$$\frac{\alpha \notin ftvs(\tau) \qquad \tau \neq \alpha'}{\vdash unify(\tau,\ \alpha) = \{\alpha\ \mapsto\ \tau\}} \tag{4.14}$$

Comment: (Rule 4.14) The condition $\tau \neq \alpha'$ ensures that only Rule 4.13 may be applied when both types are (distinct) type variables.

$$\begin{array}{lcl} ftvs(tn) & = & \emptyset \\ ftvs(tn(\overline{\tau}^k)) & = & ftvs(\overline{\tau}^k) \\ ftvs(\overline{\tau}^k) & = & ftvs(\tau^1) \cup \cdots \cup ftvs(\tau^k) \\ ftvs(\tau_1 \rightarrow \tau_2) & = & ftvs(\tau_1) \cup ftvs(\tau_2) \\ ftvs(\alpha) & = & \{\alpha\} \end{array} \tag{4.15}$$

Comment: (Rule 4.15) The $ftvs(\tau)$ function returns the set of all type variables which occur in τ.

Algorithm $\mathcal{W}$ takes the form $\mathcal{W}(\Gamma,\ expr) = (\sigma,\ \mathcal{S})$, where Γ is the typing environment, *expr* is an expression, σ is the resulting principal type of the expression, and $\mathcal{S}$ is a substitution. A definition of the algorithm for a small typed lambda language, together with a proof of soundness, is given in [Mil78] (completeness of the algorithm was proven some years later). A more modern presentation of the algorithm, as a collection of inference rules, appears in [Tof88]. In the following section we adapt these inference rules for the core Dynamic ML language.

Informally, algorithm $\mathcal{W}$ works as follows. If we examine the static semantics of Standard ML [MTHM97], it is clear that the type of each expression, even though it may not be unique, is constrained by three factors: the form of the expression, the types of any subexpressions, and the context (typing environment). The algorithm proceeds by performing a traversal of the structure of the expression. If the corresponding rule of the static semantics requires a non-deterministic choice of type, then a fresh type variable is introduced to represent this choice. If the rule also requires an equation to hold between types, the algorithm attempts to solve the equation by unification, producing a substitution. Since all of the equational constraints must hold, the substitutions are propagated (by applying them to the context) for the subexpressions, and composed in the results. The final result is the type from the static semantics, together with the most general substitution.

We define type inference for the core Dynamic ML language (as defined in Figure 4.2) by a collection of judgements shown in Figure 4.3. Each of these judgements is an implementation of Algorithm $\mathcal{W}$ for a different syntactic category of the language. Note that the results of these judgements are $\mathcal{M}\Lambda$ types and environments.

$$\begin{array}{lr}
\Gamma \vdash \mathit{program} : (\Gamma',\ \mathcal{S}) & \text{(program)} \\
\Gamma \vdash \mathit{dec} : (\Gamma',\ \mathcal{S}) & \text{(declaration)} \\
\Gamma \vdash \mathit{expr} : (\tau,\ \mathcal{S}) & \text{(expression)} \\
\Gamma \vdash \mathit{match} : (\tau,\ \mathcal{S}) & \text{(match rule)} \\
\Gamma \vdash \mathit{pat} : (\tau,\ \mathit{VE},\ \mathcal{S}) & \text{(pattern)}
\end{array}$$

Figure 4.3: *Type Inference Judgements.*

The judgements shown above collectively form the type inference algorithm. We will now provide a definition for each of these judgements in turn. As there are a large number of similar rules, we will restrict our explanation to a series of comments. To gain a better understanding of the rules, the reader will find it helpful to compare the rules with their equivalent typing rules in the definition [MTHM97]. A number of operations in these rules are defined in Section 4.3.2 where we discuss polymorphism in more detail. For convenience, we treat the environment Γ as a set in a number of rules. The set-theoretic operations on environments have the usual meanings.

Program $\Gamma \vdash \mathit{program} : (\Gamma',\ \mathcal{S})$

The elaboration process begins with Rule 4.16. A program is represented by a single declaration. Thus, we begin the judgement for the elaboration of the declaration *dec*. The resulting environment Γ' will contain only the declarations that are visible at the top-level of the program. By contrast, substitutions are not scoped, and $\mathcal{S}$ will contain a mapping for every type variable in the program to its corresponding (most general) type.

$$\frac{\Gamma \vdash \mathit{dec} : (\Gamma',\ \mathcal{S})}{\Gamma \vdash \mathit{dec} : (\Gamma',\ \mathcal{S})} \tag{4.16}$$

Declarations $\Gamma \vdash \mathit{dec} : (\Gamma',\ \mathcal{S})$

The elaboration of a declaration *dec* yields a new environment Γ' and a substitution $\mathcal{S}$. The new environment Γ' contains the elaboration environment Γ, together with any new variable, datatype, or exception declarations, and their types.

$$\frac{\begin{array}{l}\Gamma \vdash \mathit{pat} : (\tau_1,\ \mathit{VE},\ \mathcal{S}_1) \qquad \mathit{subst}(\mathcal{S}_1,\ \Gamma) \vdash \mathit{expr} : (\tau_2,\ \mathcal{S}_2) \\ \vdash \mathit{unify}(\mathit{subst}(\mathcal{S}_1\mathcal{S}_2,\ \tau_1),\ \tau_2) = \mathcal{S}_3 \qquad \mathcal{S}_4 = \mathcal{S}_3\mathcal{S}_2\mathcal{S}_1\end{array}}{\Gamma \vdash \texttt{val ()}\ \mathit{pat}\ \texttt{=}\ \mathit{expr} : (\mathit{subst}(\mathcal{S}_4,\ \Gamma \cup \mathit{VE}),\ \mathcal{S}_4)} \tag{4.17}$$

Comment: (Rule 4.17) The elaboration of a monomorphic value declaration is performed in a number of stages. The pattern *pat* is elaborated to give a type τ_1, a variable environment VE_1 containing any variables declared in *pat*, and a substitution $\mathcal{S}_1$. The expression *expr* is then elaborated to give the type τ_2, and a substitution $\mathcal{S}_2$. The types τ_1 and τ_2 are then unified with a substitution $\mathcal{S}_3$. Finally, the environment Γ is extended with VE_1 and the substitutions are composed in the result.

$$\frac{\begin{array}{l}\Gamma \vdash \mathit{tyvar}_1 : \alpha_1 \quad \cdots \quad \Gamma \vdash \mathit{tyvar}_n : \alpha_n \qquad \mathit{TE} = \{\alpha_1,\ \ldots,\ \alpha_n\} \\ \Gamma' = \Gamma \cup \mathit{TE} \qquad \Gamma' \vdash \mathit{pat} : (\tau_1,\ \mathit{VE}_1,\ \mathcal{S}_1) \qquad \mathit{subst}(\mathcal{S}_1,\ \Gamma') \vdash \mathit{expr} : (\tau_2,\ \mathcal{S}_2) \\ \vdash \mathit{unify}(\mathit{subst}(\mathcal{S}_1\mathcal{S}_2,\ \tau_1),\ \tau_2) = \mathcal{S}_3 \qquad \mathcal{S}_4 = \mathcal{S}_3\mathcal{S}_2\mathcal{S}_1 \\ \Gamma \vdash \mathit{close}(\mathit{subst}(\mathcal{S}_4,\ \mathit{VE}_1),\ \mathit{expr}) = \mathit{VE}_2 \qquad \mathit{TE} \cap \mathit{ftvs}(\mathit{VE}_1) = \emptyset\end{array}}{\Gamma \vdash \texttt{val (}\mathit{tyvar}_1\texttt{,}\ \cdots\texttt{,}\ \mathit{tyvar}_n\texttt{)}\ \mathit{pat}\ \texttt{=}\ \mathit{expr} : (\mathit{subst}(\mathcal{S}_4,\ \Gamma \cup \mathit{VE}_2),\ \mathcal{S}_4)} \tag{4.18}$$

Comment: (Rule 4.18) The elaboration of a polymorphic value declaration begins with the creation of an environment Γ' containing the type variables. The elaboration then proceeds as in the monomorphic case. However, the resulting variable environment VE_1 will contain types, rather than type-schemes. Therefore, the closure VE_2 is obtained which converts the types into general type-schemes.

$$\frac{\begin{array}{l}\Gamma \vdash \mathit{pat} : (\tau_1,\ \mathit{VE}_1,\ \mathcal{S}_1) \qquad \mathit{subst}(\mathcal{S}_1,\ \Gamma \cup \mathit{VE}_1) \vdash \mathit{expr} : (\tau_2,\ \mathcal{S}_2)\\ \vdash \mathit{unify}(\mathit{subst}(\mathcal{S}_1\mathcal{S}_2,\ \tau_1),\ \tau_2) = \mathcal{S}_3 \qquad \mathcal{S}_4 = \mathcal{S}_3\mathcal{S}_2\mathcal{S}_1\end{array}}{\Gamma \vdash \texttt{val () rec}\ \mathit{pat}\ \texttt{=}\ \mathit{expr} : (\mathit{subst}(\mathcal{S}_4,\ \Gamma \cup \mathit{VE}_1),\ \mathcal{S}_4)} \tag{4.19}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathit{tyvar}_1 : \alpha_1\ \cdots\ \Gamma \vdash \mathit{tyvar}_n : \alpha_n \qquad \mathit{TE} = \{\alpha_1,\ \ldots,\ \alpha_n\} \qquad \Gamma' = \Gamma \cup \mathit{TE}\\ \Gamma' \vdash \mathit{pat} : (\tau_1,\ \mathit{VE}_1,\ \mathcal{S}_1) \qquad \mathit{subst}(\mathcal{S}_1,\ \Gamma' \cup \mathit{VE}_1) \vdash \mathit{expr} : (\tau_2,\ \mathcal{S}_2)\\ \vdash \mathit{unify}(\mathit{subst}(\mathcal{S}_1\mathcal{S}_2,\ \tau_1),\ \tau_2) = \mathcal{S}_3 \qquad \mathcal{S}_4 = \mathcal{S}_3\mathcal{S}_2\mathcal{S}_1\\ \Gamma \vdash \mathit{close}(\mathit{subst}(\mathcal{S}_4,\ \mathit{VE}_1),\ \mathit{expr}) = \mathit{VE}_2 \qquad \mathit{TE} \cap \mathit{ftvs}(\mathit{VE}_1) = \emptyset\end{array}}{\Gamma \vdash \texttt{val (}\mathit{tyvar}_1\texttt{,}\ \cdots\texttt{,}\ \mathit{tyvar}_n\texttt{) rec}\ \mathit{pat}\ \texttt{=}\ \mathit{expr} : (\mathit{subst}(\mathcal{S}_4,\ \Gamma \cup \mathit{VE}_2),\ \mathcal{S}_4)} \tag{4.20}$$

Comment: (Rules 4.19 and 4.20) The elaboration of a recursive value declaration is performed in the same manner as a non-recursive binding. However, because the expression *expr* may refer to any of the declarations in *pat*, the environment is extended with the value environment VE_1 during the elaboration of *expr*.

$$\frac{\Gamma \vdash \mathit{dec}_1 : (\mathcal{S}_1,\ \Gamma') \qquad \mathit{subst}(\mathcal{S}_1,\ \Gamma') \vdash \mathit{dec}_2 : (\Gamma'',\ \mathcal{S}_2)}{\Gamma \vdash \mathit{dec}_1\ \langle\texttt{;}\rangle\ \mathit{dec}_2 : (\mathit{subst}(\mathcal{S}_2,\ \Gamma''),\ \mathcal{S}_2\mathcal{S}_1)} \tag{4.21}$$

Comment: (Rule 4.21) Sequentially composed declarations are evaluated in left-to-right order. The resulting environment Γ'' contains the definitions from both dec_1 and dec_2.

$$\frac{\Gamma \vdash \mathit{dec}_1 : (\Gamma',\ \mathcal{S}_1) \qquad \mathit{subst}(\mathcal{S}_1,\ \Gamma') \vdash \mathit{dec}_2 : (\Gamma'',\ \mathcal{S}_2)}{\Gamma \vdash \texttt{local}\ \mathit{dec}_1\ \texttt{in}\ \mathit{dec}_2\ \texttt{end} : (\mathit{subst}(\mathcal{S}_2,\ \Gamma'' - \Gamma'),\ \mathcal{S}_2\mathcal{S}_1)} \tag{4.22}$$

Comment: (Rule 4.22) The rule for local declarations is very similar to sequential composition. However, the resulting environment contains only the definitions from dec_2.

$$\frac{\begin{array}{l}\mathit{vid} \rightsquigarrow \mathit{con} \qquad \langle \Gamma \vdash \mathit{ty} : \tau \rangle\\ \Gamma' = (\emptyset,\ \emptyset,\ \{\mathrm{t\_exn}\ \mapsto\ (0,\ \{\mathit{con}\})\},\ \{\mathit{con}\ \mapsto\ \langle \tau \rightarrow \rangle\ \mathrm{t\_exn}\})\end{array}}{\Gamma \vdash \texttt{exception}\ \mathit{vid}\ \langle \texttt{of}\ \mathit{ty} \rangle : (\Gamma \cup \Gamma',\ \emptyset)} \tag{4.23}$$

Comment: (Rule 4.23) Dynamic ML exception identifiers *vid* are directly equivalent to $\mathcal{M}\Lambda$ exception constructors *con*. Therefore, a direct translation is performed: $\mathit{vid} \rightsquigarrow \mathit{con}$. Exceptions may be typed, though they may not be polymorphic. Exceptions in $\mathcal{M}\Lambda$ are represented as constructors of a special datatype t_exn. In the remainder of the rule, we construct a new environment Γ' containing a datatype representing the exception and combine it with Γ.

$$\frac{\Gamma \vdash \mathit{datbind} : (\mathit{DE},\ \mathit{CE})}{\Gamma \vdash \texttt{datatype}\ \mathit{datbind} : (\Gamma \cup (\mathit{DE},\ \mathit{CE}),\ \emptyset)} \tag{4.24}$$

Comment: (Rule 4.24) Datatype declarations are more complex than exception declarations, since each *datbind* may define multiple datatypes, and datatypes can be polymorphic. Hence, a separate judgement is used $\Gamma \vdash \mathit{datbind} : (\mathit{DE},\ \mathit{CE})$. This judgement does not return a substitution since no type inference is necessary. The environments *DE* and *CE* contain the datatypes and constructors respectively.

Datatype Bindings $\Gamma \vdash \mathit{datbind} : (\mathit{DE},\ \mathit{CE})$

Each datatype binding *datbind* may contain a sequence of datatype declarations. Polymorphic and monomorphic datatype declarations may also be combined in a single sequence. Therefore, datatype declarations are processed by the following doubly-recursive rules. Rule 4.25 is used for monomorphic declarations and Rule 4.26 is used for polymorphic declarations. The rules are essentially the same: the constructors of a single datatype are converted into a constructor environment by a further judgement $\Gamma,\ \mathit{tn}\ \langle,\ \overline{\alpha}^k\rangle \vdash \mathit{conbind} : CE$. The names of the constructors (obtained from the domain of the constructor environment Dom CE_1) are then recorded in a new datatype environment, together with the arity of the constructor (the number of type variables). Finally, the datatype environments of any remaining datatypes in the sequence are combined, as are the constructor environments. The result is a single datatype environment containing all of the datatype declarations, and a single constructor environment containing all of the constructor declarations.

$$\frac{\langle \Gamma \vdash \mathit{datbind} : (\mathit{DE}_1,\ \mathit{CE}_1)\rangle \qquad \mathit{tycon} \rightsquigarrow \mathit{tn} \qquad \Gamma,\ \mathit{tn} \vdash \mathit{conbind} : \mathit{CE}_2}{\begin{array}{l}\Gamma \vdash \texttt{()}\ \mathit{tycon}\ \texttt{=}\ \mathit{conbind}\ \langle \texttt{and}\ \mathit{datbind}\rangle : \\ \quad (\{\mathit{tn}\ \mapsto\ (0,\ \mathrm{Dom}\ \mathit{CE}_1)\}\ \langle \cup\ \mathit{DE}_1\rangle,\ \mathit{CE}_2\ \langle \cup\ \mathit{CE}_1\rangle)\end{array}} \tag{4.25}$$

$$\frac{\begin{array}{l}\langle \Gamma \vdash \mathit{datbind} : (\mathit{DE}_1,\ \mathit{CE}_2)\rangle \qquad \mathit{tycon} \rightsquigarrow \mathit{tn} \\ \Gamma \vdash \mathit{tvar}_1 : \alpha^1 \quad \cdots \quad \Gamma \vdash \mathit{tvar}_n : \alpha^n \qquad \Gamma,\ \mathit{tn},\ \overline{\alpha}^n \vdash \mathit{conbind} : \mathit{CE}_1\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{(}\mathit{tyvar}_1\texttt{,}\ \cdots\texttt{,}\ \mathit{tyvar}_n\texttt{)}\ \mathit{tycon}\ \texttt{=}\ \mathit{conbind}\ \langle \texttt{and}\ \mathit{datbind}\rangle : \\ \quad (\{\mathit{tn}\ \mapsto\ (n,\ \mathrm{Dom}\ \mathit{CE}_1)\}\ \langle \cup\ \mathit{DE}_1\rangle,\ \mathit{CE}_1\ \langle \cup\ \mathit{CE}_2\rangle)\end{array}} \tag{4.26}$$

Comment: (Rules 4.25 and 4.26) These rules construct the environment which is used in the definition of Rules 4.5 and 4.6.

Constructor Bindings $\Gamma,\ tn\ \langle,\ \overline{\alpha}^k\rangle \vdash conbind : CE$

Constructor bindings *conbind* are translated into constructor environments in Rule 4.27 below. A single *conbind* may contain a sequence of constructors. Therefore, as with the *datbind* rules, a recursive definition is used. However, note that monomorphic and polymorphic constructors may not be combined in a single sequence. A sequence of mappings between the constructor names *con* and their types are created to form the constructor environment *CE*. The *ctype* function (4.28) is used to determine the type of each constructor. There are four cases, corresponding to nullary monomorphic, unary monomorphic, nullary polymorphic and unary polymorphic constructors respectively.

$$\frac{vid \rightsquigarrow con \qquad \langle\langle \Gamma \vdash ty : \tau \rangle\rangle \qquad \langle\langle\langle \Gamma,\ tn\ \langle,\ \overline{\alpha}^k\rangle \vdash conbind : CE \rangle\rangle\rangle}{\begin{array}{l}\Gamma,\ tn\ \langle,\ \overline{\alpha}^k\rangle \vdash vid\ \langle\langle \texttt{of}\ ty\rangle\rangle\ \langle\langle\langle \texttt{|}\ conbind\rangle\rangle\rangle : \\ \quad \{con\ \mapsto\ ctype(tn\ \langle,\ \overline{\alpha}^k\rangle\ \langle\langle,\ \tau_1\rangle\rangle)\}\ \langle\langle\langle \cup\ CE\rangle\rangle\rangle\end{array}} \tag{4.27}$$

$$\begin{array}{lcl} ctype(tn) & = & tn \\ ctype(tn,\ \tau) & = & \tau \to tn \\ ctype(tn,\ \overline{\alpha}^k) & = & \forall\, \overline{\alpha}^k.\ tn(\overline{\alpha}^k) \\ ctype(tn,\ \overline{\alpha}^k,\ \tau) & = & \forall\, \overline{\alpha}^k.\ \tau \to tn(\overline{\alpha}^k) \end{array} \tag{4.28}$$

Expressions $\Gamma \vdash expr : (\tau,\ \mathcal{S})$

The elaboration of an expression *expr* results in a type τ and a substitution $\mathcal{S}$. As in $\mathcal{M}\Lambda$, the expressions all have monomorphic types. This is because Dynamic ML only contains let polymorphism.

$$\frac{}{\Gamma \vdash scon : (tn_{scon},\ \emptyset)} \tag{4.29}$$

$$\frac{vid \rightsquigarrow x \qquad \Gamma(x) = \tau}{\Gamma \vdash vid : (\tau,\ \emptyset)} \tag{4.30}$$

$$\frac{\begin{array}{l} vid \rightsquigarrow x \qquad \Gamma(x) = \forall\, \overline{\alpha_1}^k.\ \tau_1 \qquad \overline{\alpha_2}^k \text{ fresh} \\ subst(\{\alpha_1^1\ \mapsto\ \alpha_2^1,\ \ldots,\ \alpha_1^k\ \mapsto\ \alpha_2^k\},\ \tau_1) = \tau_2 \end{array}}{\Gamma \vdash vid : (\tau_2,\ \emptyset)} \tag{4.31}$$

Comment: (Rules 4.30 and 4.31) The type of an identifier is obtained directly from the environment. If the type is polymorphic (Rule 4.31), then the type variables are substituted with fresh type variables. This allows multiple instances of an identifier to have different types.

$$\frac{}{\Gamma \vdash \texttt{\{\}} : (\text{t\_unit},\ \emptyset)} \tag{4.32}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathit{expr}_1 : (\tau_1,\ \mathcal{S}_1) \qquad \mathit{subst}(\mathcal{S}_1,\ \Gamma) \vdash \mathit{expr}_2 : (\tau_2,\ \mathcal{S}_2)\ \cdots \\ \quad \mathit{subst}(\mathcal{S}_1 \cdots \mathcal{S}_{(n-1)},\ \Gamma) \vdash \mathit{expr}_n : (\tau_n,\ \mathcal{S}_n) \\ \delta \text{ a permutation of } 1 \ldots n, \text{ where } \mathit{lab}_{\delta(1)} < \cdots < \mathit{lab}_{\delta(n)}\end{array}}{\Gamma \vdash \texttt{\{}\mathit{lab}_1 \texttt{ = } \mathit{expr}_1\texttt{,}\ \cdots\texttt{,}\ \mathit{lab}_n \texttt{ = } \mathit{expr}_n\texttt{\}} : ((\tau_{\delta(1)},\ \ldots,\ \tau_{\delta(n)}),\ \mathcal{S}_1 \cdots \mathcal{S}_n)} \tag{4.33}$$

Comment: (Rule 4.33) Dynamic ML records are represented by unlabelled $\mathcal{M}\Lambda$ tuples. Therefore, as in Rule 4.4, we permute the resulting types by the function δ, which defines a lexicographic ordering of the labels.

$$\frac{\Gamma \vdash \mathit{dec} : (\Gamma',\ \mathcal{S}_1) \qquad \mathit{subst}(\mathcal{S}_1,\ \Gamma') \vdash \mathit{expr} : (\tau,\ \mathcal{S}_2)}{\Gamma \vdash \texttt{let}\ \mathit{dec}\ \texttt{in}\ \mathit{expr}\ \texttt{end} : (\tau,\ \mathcal{S}_2\mathcal{S}_1)} \tag{4.34}$$

Comment: (Rule 4.34) In a `let` expression, the declaration *dec* is elaborated to provide an environment Γ' used in the elaboration of the expression *expr*. This environment is not preserved at the end of the rule, correctly limiting the scope of the declaration. Note that the type of the whole expression is the type τ of the subexpression.

$$\frac{\begin{array}{l}\Gamma \vdash \mathit{expr}_1 : (\tau_1,\ \mathcal{S}_1) \qquad \mathit{subst}(\mathcal{S}_1,\ \Gamma) \vdash \mathit{expr}_2 : (\tau_2,\ \mathcal{S}_2) \\ \alpha \text{ fresh} \qquad \vdash \mathit{unify}(\mathit{subst}(\mathcal{S}_2,\ \tau_1),\ \tau_2 \to \alpha) = \mathcal{S}_3\end{array}}{\Gamma \vdash \mathit{expr}_1\ \mathit{expr}_2 : (\mathit{subst}(\mathcal{S}_3,\ \alpha),\ \mathcal{S}_3\mathcal{S}_2\mathcal{S}_1)} \tag{4.35}$$

Comment: (Rule 4.35) The application of a function expression expr_1 to an argument expression expr_2 is captured in this rule. The unification is essentially a check that τ_1 is a function type. Note that a fresh type variable α is introduced in order that we may return the result type of the function at the end of the rule.

$$\frac{\Gamma \vdash \mathit{expr} : (\tau_1,\ \mathcal{S}_1) \qquad \mathit{subst}(\mathcal{S}_1,\ \Gamma) \vdash \mathit{ty} : \tau_2 \qquad \vdash \mathit{unify}(\tau_1,\ \tau_2) = \mathcal{S}_2}{\Gamma \vdash (\mathit{expr}\ :\ \mathit{ty}) : (\mathit{subst}(\mathcal{S}_2,\ \tau_2),\ \mathcal{S}_1\mathcal{S}_2)} \tag{4.36}$$

Comment: (Rule 4.36) A type constraint may restrict the polymorphism of the expression. By unifying the type constraint with the type of the expression, we obtain the appropriate substitution.

$$\frac{\begin{array}{l}\Gamma \vdash \mathit{expr} : (\tau_1,\ \mathcal{S}_1) \qquad \mathit{subst}(\mathcal{S}_1,\ \Gamma) \vdash \mathit{match} : (\tau_2,\ \mathcal{S}_2) \\ \vdash \mathit{unify}(\text{t\_exn} \to \tau_1,\ \tau_2) = \mathcal{S}_3\end{array}}{\Gamma \vdash \mathit{expr}\ \texttt{handle}\ \mathit{match} : (\mathit{subst}(\mathcal{S}_3\mathcal{S}_2,\ \tau_1),\ \mathcal{S}_3\mathcal{S}_2\mathcal{S}_1)} \tag{4.37}$$

Comment: (Rule 4.37) In this rule, we check that the exception handler *match* is a function with the correct argument type t_exn.

$$\frac{\Gamma \vdash \mathit{expr} : (\tau_1,\ \mathcal{S}_1) \qquad \vdash \mathit{unify}(\text{t\_exn},\ \tau_1) = \mathcal{S}_2 \qquad \alpha \text{ fresh}}{\Gamma \vdash \texttt{raise}\ \mathit{expr} : (\alpha,\ \mathcal{S}_2\mathcal{S}_1)} \tag{4.38}$$

Comment: (Rule 4.38) Raising an exception corresponds to applying an exception handling function. We check that the exception expression *expr* has the correct type t_exn for the handle function. A fresh type variable is introduced as the type of the whole expression, since there is no other appropriate type.

$$\frac{\Gamma \vdash \mathit{match} : (\tau_1,\ \mathcal{S}_1)}{\Gamma \vdash \texttt{fn}\ \mathit{match} : (\tau_1,\ \mathcal{S}_1)} \tag{4.39}$$

Comment: (Rule 4.39) The body of a Dynamic ML function is defined by a *match*, consisting of a sequence of *pat* `=>` *expr* pairs.

Match Rules $\Gamma \vdash \mathit{match} : (\tau,\ \mathcal{S})$

A *match* takes the form of a function with a sequence of patterns as arguments, and a corresponding sequence of expressions as results. The elaboration of a *match* is performed recursively in left-right order. The base-case, a single pattern/expression pair, is defined by Rule 4.40. The recursive-case, for a sequence of patterns and expressions, is defined by Rule 4.41. We must check that every pattern/expression pair has the correct type, i.e. they can all be unified to a single function type.

$$\frac{\Gamma \vdash \mathit{pat} : (\tau_1,\ \mathit{VE}_1,\ \mathcal{S}_1) \qquad \mathit{subst}(\mathcal{S}_1,\ \Gamma) \cup \mathit{VE}_1 \vdash \mathit{expr} : (\mathcal{S}_2,\ \tau_2)}{\Gamma \vdash \mathit{pat}\ \texttt{=>}\ \mathit{expr} : (\mathit{subst}(\mathcal{S}_2\mathcal{S}_1,\ \tau_1) \to \tau_2,\ \mathcal{S}_2\mathcal{S}_1)} \tag{4.40}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathit{pat} : (\tau_1,\ \mathit{VE}_1,\ \mathcal{S}_1) \qquad \mathit{subst}(\mathcal{S}_1,\ \Gamma) \cup \mathit{VE}_1 \vdash \mathit{expr} : (\mathcal{S}_2,\ \tau_2) \\ \mathit{subst}(\mathcal{S}_2\mathcal{S}_1,\ \Gamma) \vdash \mathit{match} : (\mathcal{S}_3,\ \tau_3) \\ \vdash \mathit{unify}(\tau_3,\ \mathit{subst}(\mathcal{S}_3,\ \mathit{subst}(\mathcal{S}_2\mathcal{S}_1,\ \tau_1) \to \tau_2)) = \mathcal{S}_4\end{array}}{\Gamma \vdash \mathit{pat}\ \texttt{=>}\ \mathit{expr}\ \texttt{|}\ \mathit{match} : (\mathit{subst}(\mathcal{S}_4,\ \tau_3),\ \mathcal{S}_4\mathcal{S}_3\mathcal{S}_2\mathcal{S}_1)} \tag{4.41}$$

Patterns $\Gamma \vdash pat : (\tau,\ VE,\ \mathcal{S})$

Informally, a pattern *pat* is a description of a set of values. In the following rules, we determine the (most general) type of this set. A pattern may define variables, which will be bound on a successful match. Thus, in addition to the type τ and substitution $\mathcal{S}$, the elaboration of a pattern also results in a variable environment *VE*. The variable environment maps any variables bound in a pattern to their corresponding types.

$$\frac{\alpha \text{ fresh}}{\Gamma \vdash \texttt{\_} : (\alpha,\ \emptyset,\ \emptyset)} \tag{4.42}$$

Comment: (Rule 4.42) The type of the wild-card pattern is unknown, hence a fresh type variable is introduced.

$$\frac{}{\Gamma \vdash scon : (tn_{scon},\ \emptyset,\ \emptyset)} \tag{4.43}$$

$$\frac{vid \rightsquigarrow x \qquad x \notin \mathrm{Dom}\,\Gamma \qquad \alpha \text{ fresh}}{\Gamma \vdash vid : (\alpha,\ \{x \mapsto \alpha\},\ \emptyset)} \tag{4.44}$$

Comment: (Rule 4.44) The condition $x \notin \mathrm{Dom}\,\Gamma$ ensures that a single variable cannot occur more than once in a pattern.

$$\frac{vid \rightsquigarrow con \qquad \Gamma(con) = \tau}{\Gamma \vdash vid : (\tau,\ \emptyset,\ \emptyset)} \tag{4.45}$$

$$\frac{\begin{array}{l} vid \rightsquigarrow con \qquad \Gamma(con) = \forall\, \overline{\alpha_1}^k .\ \tau_1 \qquad \overline{\alpha_2}^k \text{ fresh} \\ subst(\{\alpha_1^1 \mapsto \alpha_2^1,\ \ldots,\ \alpha_1^k \mapsto \alpha_2^k\},\ \tau_1) = \tau_2 \end{array}}{\Gamma \vdash vid : (\tau_2,\ \emptyset,\ \emptyset)} \tag{4.46}$$

Comment: (Rules 4.45 and 4.46) The type of a constructor which occurs in a pattern is obtained from the environment. If the constructor is polymorphic, its type variables are mapped to a fresh set.

$$\frac{}{\Gamma \vdash \texttt{\{\}} : (\text{t\_unit},\ \emptyset,\ \emptyset)} \tag{4.47}$$

$$\frac{\begin{array}{l} \Gamma \vdash pat_1 : (\tau_1,\ VE_1,\ \mathcal{S}_1) \quad \cdots \quad \Gamma \vdash pat_n : (\tau_n,\ VE_n,\ \mathcal{S}_n) \\ \delta \text{ a permutation of } 1 \ldots n, \text{ where } lab_{\delta(1)} < \cdots < lab_{\delta(n)} \end{array}}{\begin{array}{l} \Gamma \vdash \texttt{\{}lab_1 \texttt{=} pat_1 \texttt{,}\ \cdots \texttt{,}\ lab_n \texttt{=} pat_n\texttt{\}} : \\ \quad ((\tau_{\delta(1)},\ \ldots,\ \tau_{\delta(n)}),\ VE_1 \cup \cdots \cup VE_n,\ \mathcal{S}_1 \cdots \mathcal{S}_n) \end{array}} \tag{4.48}$$

$$\frac{\begin{array}{l}\Gamma \vdash pat : (\tau_1,\ VE_1,\ \mathcal{S}_1) \qquad \Gamma \vdash ty : \tau_2 \\ \vdash unify(\tau_1,\ \tau_2) = \mathcal{S}_2 \qquad \mathcal{S}_3 = \mathcal{S}_2\mathcal{S}_1\end{array}}{\Gamma \vdash (pat\ :\ ty) : (subst(\mathcal{S}_3,\ \tau_1),\ subst(\mathcal{S}_3,\ VE_1),\ \mathcal{S}_3)} \tag{4.49}$$

Comment: (Rule 4.49) The type *ty* may restrict the type of the pattern. Thus, a unification is performed between the pattern type τ_1 and the type constraint τ_2.

$$\frac{\begin{array}{l}vid \rightsquigarrow con \qquad \Gamma \vdash pat : (\tau_1,\ VE_1,\ \mathcal{S}_1) \\ \Gamma(con) = \forall\, \overline{\alpha_1}^k.\ \tau_2 \qquad \overline{\alpha_2}^k \text{ fresh} \\ subst(\{\alpha_1^1 \mapsto \alpha_2^1,\ \ldots,\ \alpha_1^k \mapsto \alpha_2^k\},\ \tau_2) = \tau_3 \\ \alpha_3 \text{ fresh} \qquad \vdash unify(\tau_1 \to \alpha_3,\ \tau_3) = \mathcal{S}_2\end{array}}{\Gamma \vdash vid\, pat : (subst(\mathcal{S}_2,\ \alpha_3),\ subst(\mathcal{S}_2,\ VE_1),\ \mathcal{S}_2\mathcal{S}_1)} \tag{4.50}$$

Comment: (Rule 4.50) A constructor is represented as a function. Therefore, the type τ_1 of the constructed pattern *pat* must match the argument type τ_3 of the constructor function.

$$\frac{\begin{array}{l}vid \rightsquigarrow x \qquad \Gamma \vdash pat : (\tau_1,\ VE_1,\ \mathcal{S}_1) \qquad x \notin \text{Dom } VE_1 \\ \langle \Gamma \vdash ty : \tau_2 \quad \vdash unify(\tau_1,\ \tau_2) = \mathcal{S}_2 \rangle \qquad \mathcal{S}_3 = \mathcal{S}_1\langle \mathcal{S}_2 \rangle\end{array}}{\Gamma \vdash vid\ \langle :\ ty\rangle\ \texttt{as}\ pat : (subst(\mathcal{S}_3,\ \tau_1),\ subst(\mathcal{S}_3,\ VE_1 \cup \{x \mapsto \tau_1\}),\ \mathcal{S}_3)} \tag{4.51}$$

Comment: (Rule 4.51) The test $x \notin$ Dom VE_1 is required to ensure that the variable x does not occur inside the layered pattern *pat*.

4.3.2 Let-bound Polymorphism

The judgement for the elaboration of expressions $\Gamma \vdash expr : (\tau,\ \mathcal{S})$ results in a type τ rather than a general type scheme σ as we would expect from algorithm $\mathcal{W}$. This is because of let polymorphism in Dynamic ML. Type schemes are only introduced inside `val` and `val rec` declarations (Rules 4.18 and 4.20). In these rules, types are converted into type schemes by the closure function *close* operation which we define below. Recall that a similar operation was necessary when type checking **let** and **fix** expressions in Chapter 3. The act of forming a type scheme from a type is called *quantification*. A total closure of a type τ is the type scheme $\sigma = \forall ftvs(\tau).\ \tau$. The closure operation which we now define in Rules 4.52 and 4.53, operates on a variable environment *VE*, rather than a single type.

$$\frac{\begin{array}{l}\Gamma \vdash \mathit{nexp}(\mathit{expr}) \\ \mathit{VE}_1 = \{x^1 \;\mapsto\; \tau^1,\ \ldots,\ x^k \;\mapsto\; \tau^k\} \\ \mathit{VE}_2 = \{x^1 \;\mapsto\; \forall(\mathit{ftvs}(\tau^1) - \mathit{ftvs}(\Gamma)).\,\tau^1,\ \ldots \\ \qquad\qquad \ldots,\ x^k \;\mapsto\; \forall(\mathit{ftvs}(\tau^k) - \mathit{ftvs}(\Gamma)).\,\tau^k\}\end{array}}{\Gamma \vdash \mathit{close}(\mathit{VE}_1,\ \mathit{expr}) = \mathit{VE}_2} \tag{4.52}$$

$$\frac{\Gamma \vdash \neg\mathit{nexp}(\mathit{expr})}{\Gamma \vdash \mathit{close}(\mathit{VE}_1,\ \mathit{expr}) = \mathit{VE}_1} \tag{4.53}$$

When constructing the closure of an variable environment VE, we do not construct the total closure; any type variables appearing in the environment $ftvs(\Gamma)$ are excluded (Rule 4.52). Moreover, we do not close the environments corresponding to the class of *expansive* expression (Rule 4.53). These are expressions which generate exceptions or manipulate references. The non-expansive expressions are defined by the predicate *nexp* in Figure 4.4.

A detailed justification of the closure operation is given in [MT91]. We briefly note that using the total closure may result in an unsound elaboration, as polymorphic declarations can be embedded, i.e. we do not want to capture all of the free type variables. Also, exceptions and references depend on the type in the memory rather than the type in the environment Γ, and polymorphic exceptions are forbidden.

$$\frac{}{\Gamma \vdash \mathit{nexp}(\mathit{scon})} \tag{4.54}$$

$$\frac{}{\Gamma \vdash \mathit{nexp}(\mathtt{fn}\ \mathit{match})} \tag{4.57}$$

$$\frac{}{\Gamma \vdash \mathit{nexp}(\mathit{vid})} \tag{4.55}$$

$$\frac{}{\Gamma \vdash \mathit{nexp}(\mathit{expr}\ :\ \mathit{ty})} \tag{4.58}$$

$$\frac{\Gamma \vdash \mathit{nexp}(\mathit{expr}_1\ \mathit{expr}_2)}{\Gamma \vdash \mathit{nexp}((\mathit{expr}_1\ :\ \mathit{ty})\ \mathit{expr}_2)} \tag{4.56}$$

$$\frac{\mathit{vid} \leadsto x}{\Gamma \vdash \mathit{nexp}(\mathit{vid}\ \mathit{expr})} \tag{4.59}$$

$$\frac{\mathit{vid} \leadsto \mathit{con} \qquad \vdash \mathit{dtype}(\Gamma(\mathit{con})) \neq \text{t\_ref} \qquad \vdash \mathit{dtype}(\Gamma(\mathit{con})) \neq \text{t\_exn}}{\Gamma \vdash \mathit{nexp}(\mathit{vid}\ \mathit{expr})} \tag{4.60}$$

$$\frac{\Gamma \vdash \mathit{nexp}(\mathit{expr}_1) \quad \cdots \quad \Gamma \vdash \mathit{nexp}(\mathit{expr}_n)}{\Gamma \vdash \mathit{nexp}(\{\mathit{lab}_1 = \mathit{expr}_1\ ,\ \cdots\ ,\ \mathit{lab}_n = \mathit{expr}_n\})} \tag{4.61}$$

Figure 4.4: *Non-Expansive Expressions.*

Polymorphic Operations

We have used the $ftvs$ and $subst$ operations on type schemes and environments in our earlier rules. The additional cases required to cover polymorphism are defined below. This completes the definition of the type inference algorithm for Dynamic ML.

$$\frac{}{\vdash \mathit{ftvs}(\forall \overline{\alpha}^{k}.\ \tau) = \mathit{ftvs}(\tau) - \{\alpha^1,\ \ldots,\ \alpha^k\}} \tag{4.62}$$

Comment: (Rule 4.62) The free type variables of a type scheme are those which are not captured by the quantification.

$$\frac{\mathit{VE} = \{x^1\ \mapsto\ \sigma^1,\ \ldots,\ x^k\ \mapsto\ \sigma^k\}}{\vdash \mathit{ftvs}(\mathit{VE}) = \mathit{ftvs}(\sigma^1) \cup \cdots \cup \mathit{ftvs}(\sigma^k)} \tag{4.63}$$

Comment: (Rule 4.63) The type variables for a variable environment VE are obtained from the union of all of the type variables in the range.

$$\frac{}{\vdash \mathit{ftvs}(\Gamma) = \mathit{TE} \in \Gamma} \tag{4.64}$$

Comment: (Rule 4.64) The free type variables of an environment are contained in the type environment TE

$$\frac{\mathit{subst}(\mathcal{S},\ \tau_1) = \tau_2}{\vdash \mathit{subst}(\mathcal{S},\ \forall \overline{\alpha}^{k}.\ \tau_1) = \forall ftvs(\tau_2).\ \tau_2} \tag{4.65}$$

Comment: (Rule 4.65) The resulting type scheme is constructed as a total closure after performing the substitution on the quantified type τ_1.

$$\frac{\begin{array}{l}\mathit{VE}_1 = \{x^1\ \mapsto\ \sigma^1,\ \ldots,\ x^k\ \mapsto\ \sigma^k\} \\ \mathit{VE}_2 = \{x^1\ \mapsto\ \mathit{subst}(\mathcal{S},\ \sigma^1),\ \ldots,\ x^k\ \mapsto\ \mathit{subst}(\mathcal{S},\ \sigma^k)\}\end{array}}{\vdash \mathit{subst}(\mathcal{S},\ \mathit{VE}_1) = \mathit{VE}_2} \tag{4.66}$$

Comment: (Rule 4.66) A substitution over a variable environment is achieved by performing a substitution over each of the types in the range.

$$\frac{\Gamma = (\mathit{TE},\ \mathit{VE}_1,\ \mathit{DE},\ \mathit{CE}) \qquad \vdash \mathit{subst}(\mathcal{S},\ \mathit{VE}_1) = \mathit{VE}_2}{\vdash \mathit{subst}(\mathcal{S},\ \Gamma) = (\mathit{TE},\ \mathit{VE}_2,\ \mathit{DE},\ \mathit{CE})} \tag{4.67}$$

4.4 Syntax-Directed Translation

The remaining step in the translation of Dynamic ML is the generation of the $\mathcal{M}\Lambda$ expressions. In particular, the generation of appropriate $\mathcal{M}\Lambda$ expressions for Dynamic ML pattern matching. The remaining Dynamic ML expressions are translated by a relatively straightforward process as they are largely similar to their $\mathcal{M}\Lambda$ equivalents. Only minor changes are required, e.g. hoisting the datatype and exception declarations to the top-level. The translation is defined by a set of rules shown in Figure 4.5.

$$\begin{array}{ll}
\Gamma,\ \mathcal{S} \vdash \mathit{program} \rightsquigarrow P & \text{(program)} \\
\mathcal{S},\ E_1 \vdash \mathit{dec} \rightsquigarrow (\overline{\overline{D}},\ \overline{\overline{X}},\ E_2) & \text{(declaration)} \\
\mathcal{S} \vdash \mathit{expr} \rightsquigarrow (\overline{\overline{D}},\ \overline{\overline{X}},\ E) & \text{(expression)} \\
\mathcal{S},\ E_1,\ E_2 \vdash \mathit{match} \rightsquigarrow (\overline{\overline{D}},\ \overline{\overline{X}},\ E_3) & \text{(match rule)} \\
\mathcal{S},\ E_1,\ E_2,\ E_3 \vdash \mathit{pat} \rightsquigarrow E_4 & \text{(pattern)}
\end{array}$$

Figure 4.5: *Translation Rules.*

With reference to Rule 4.1, the elaboration step produces an environment Γ' and a substitution $\mathcal{S}$, which are passed to the translation step. The environment Γ' contains all of the top-level declarations, and the substitution $\mathcal{S}$ maps the free type variables in the program to their corresponding types. After elaboration, the program will contain type annotations. We make use of these annotations in a number of rules. For example, the conclusion of Rule 4.80 reads $\mathcal{S} \vdash \mathit{vid}\ (: \tau) \rightsquigarrow (\emptyset,\ \emptyset,\ \mathbf{var}\ x)$. Here, the type τ enclosed in parentheses is the type annotated during elaboration. Note that the annotated type may contain free type variables, and therefore must be substituted from $\mathcal{S}$.

Program $\Gamma,\ \mathcal{S} \vdash \mathit{program} \rightsquigarrow P$

The translation begins with Rule 4.68; a Dynamic ML program *dec* is translated into a $\mathcal{M}\Lambda$ program $(\overline{\overline{D}},\ \overline{\overline{X}},\ E)$. The result of evaluating a program is a tuple containing all of the variables which are visible at the top-level.

$$\frac{\begin{array}{l} \mathit{VE} \in \Gamma = \{x_1 \ \mapsto\ \sigma_1,\ \ldots,\ x_k \ \mapsto\ \sigma_k\} \\ \mathcal{S},\ \mathbf{tuple}\ (\mathbf{var}\ x_1,\ \ldots,\ \mathbf{var}\ \ x_k) \vdash \mathit{dec} \rightsquigarrow (\overline{\overline{D}},\ \overline{\overline{X}},\ E) \end{array}}{\Gamma,\ \mathcal{S} \vdash \mathit{dec} \rightsquigarrow (\overline{\overline{D}},\ \overline{\overline{X}},\ E)} \qquad (4.68)$$

Declarations $\mathcal{S},\ E \vdash \mathit{dec} \rightsquigarrow P$

A Dynamic ML declaration is translated by the rule $\mathcal{S},\ E \vdash \mathit{dec} \rightsquigarrow P$. The expression E is the scope of the declaration, i.e. `let` *dec* `in` E `end`.

$$\frac{\begin{array}{l}\mathcal{S} \vdash \mathit{expr}\ (:\tau) \rightsquigarrow (\overline{\overline{D_1}},\ \overline{\overline{X_1}},\ E_2) \qquad x \text{ fresh} \\ \mathcal{S},\ E_2,\ E_1,\ \textbf{raise}\ (\text{e\_bind},\ \mathit{subst}(\mathcal{S},\ \tau)) \vdash \mathit{pat} \rightsquigarrow E_3\end{array}}{\mathcal{S},\ E_1 \vdash \texttt{val (}\mathit{tyvar}_1\texttt{,}\ \cdots\texttt{,}\ \mathit{tyvar}_n\texttt{)}\ \mathit{pat}\ \texttt{=}\ \mathit{expr} \rightsquigarrow (\overline{\overline{D_1}},\ \overline{\overline{X_1}},\ E_3)} \tag{4.69}$$

Comment: (Rule 4.69) Translating the pattern *pat* will result in a **let** expression (Section 4.4.1). Therefore we do not need to introduce one directly in this rule. An e_bind exception will be raised if the pattern is not matched. Polymorphic value declarations are translated in an identical manner to monomorphic declarations. The explicit type variables tyvar_1, $\cdots$, tyvar_n are only used during elaboration and are ignored.

$$\frac{\begin{array}{l}\mathit{vid}_1 \rightsquigarrow x_1^1 \quad \cdots \quad \mathit{vid}_n \rightsquigarrow x_1^n \qquad \overline{x_2}^{\,n} \text{ fresh} \qquad \Gamma = (\mathit{TE},\ \mathit{VE},\ \mathit{DE},\ \mathit{CE}) \\ \mathcal{S},\ \textbf{var}\ x_2^1,\ \textbf{raise}\ (\text{e\_bind},\ \mathit{ret}(\mathit{VE}(x_1^1))) \vdash \mathit{match}_1 \rightsquigarrow (\overline{\overline{D_1}},\ \overline{\overline{X_1}},\ E_2^1) \ldots \\ \quad \mathcal{S},\ \textbf{var}\ x_2^n,\ \textbf{raise}\ (\text{e\_bind},\ \mathit{ret}(\mathit{VE}(x_1^n))) \vdash \mathit{match}_n \rightsquigarrow (\overline{\overline{D_n}},\ \overline{\overline{X_n}},\ E_2^n)\end{array}}{\begin{array}{l}\mathcal{S},\ E_1 \vdash \texttt{val (}\mathit{tyvar}_1\texttt{,}\ \cdots\texttt{,}\ \mathit{tyvar}_n\texttt{) rec \{}\mathit{lab}_1\ \texttt{=}\ \mathit{vid}_1\texttt{,}\ \cdots\texttt{,}\ \mathit{lab}_n\ \texttt{=}\ \mathit{vid}_n\texttt{\}}\ \texttt{=} \\ \quad \texttt{\{}\mathit{lab}_1\ \texttt{= fn}\ \mathit{match}_1\texttt{,}\ \cdots\texttt{,}\ \mathit{lab}_n\ \texttt{= fn}\ \mathit{match}_n\texttt{\}}\ (:\Gamma) \rightsquigarrow \\ \qquad (\overline{\overline{D_1}} \cup \cdots \cup \overline{\overline{D_n}},\ \overline{\overline{X_1}} \cup \cdots \cup \overline{\overline{X_n}}, \\ \qquad\quad \textbf{fix}\ ((x_1^1,\ \mathit{VE}(x_1^1)) = (\textbf{fn}\ (x_2^1,\ \mathit{VE}(x_1^1))\ =\ E_2^1),\ \ldots \\ \qquad\qquad \ldots,\ (x_1^n,\ \mathit{VE}(x_1^n)) = (\textbf{fn}\ (x_1^n,\ \mathit{VE}(x_1^n))\ =\ E_2^n))\ \textbf{in}\ E_1\end{array}} \tag{4.70}$$

Comment: (Rule 4.70) A `val rec` declaration in Dynamic ML, containing a sequence of recursive function declarations, is translated into an $\mathcal{M}\Lambda$ **fix** expression. The names of the functions are given by $\mathit{vid}_1,\ \ldots,\ \mathit{vid}_n$. The first step is to translate these names into their $\mathcal{M}\Lambda$ counterparts $x_1^1,\ \ldots,\ x_1^n$. We also introduce a fresh set of variables $\overline{x_2}^{\,n}$ which will hold the arguments to each function. The type of each function is obtained from *VE* in the annotated environment Γ. Polymorphic functions will have already elaborated to type schemes in *VE*. The function expressions $\mathit{match}_1,\ \ldots,\ \mathit{match}_n$ are translated in-turn into $\mathcal{M}\Lambda$ expressions $E_2^1,\ \ldots,\ E_2^n$. Each function may raise an e_bind exception, if an incomplete match is specified. The type of this exception is the return type of the function, where $\mathit{ret}(\tau_1 \rightarrow \tau_2) = \tau_2$. The translated functions E_2 are gathered together into a large **fix** expression at the end of the rule. Any datatypes or exceptions which are defined within any function are hoisted to the top-level. Although the functions are supplied as a record, we do not sort them by name here, as the order in which they occur in the **fix** expression is unimportant.

$$\frac{\mathcal{S},\ E_1 \vdash \mathit{dec}_2 \leadsto (\overline{\overline{D_1}},\ \overline{\overline{X_1}},\ E_2) \qquad \mathcal{S},\ E_2 \vdash \mathit{dec}_1 \leadsto (\overline{\overline{D_2}},\ \overline{\overline{X_2}},\ E_3)}{\mathcal{S},\ E_1 \vdash \mathit{dec}_1\ \langle\texttt{;}\rangle\ \mathit{dec}_2 \leadsto (\overline{\overline{D_1}} \cup \overline{\overline{D_2}},\ \overline{\overline{X_1}} \cup \overline{\overline{X_2}},\ E_3)} \tag{4.71}$$

$$\frac{\mathcal{S},\ E_1 \vdash \mathit{dec}_2 \leadsto (\overline{\overline{D_1}},\ \overline{\overline{X_1}},\ E_2) \qquad \mathcal{S},\ E_2 \vdash \mathit{dec}_1 \leadsto (\overline{\overline{D_2}},\ \overline{\overline{X_2}},\ E_3)}{\mathcal{S},\ E_1 \vdash \texttt{local}\ \mathit{dec}_1\ \texttt{in}\ \mathit{dec}_2\ \texttt{end} \leadsto (\overline{\overline{D_1}} \cup \overline{\overline{D_2}},\ \overline{\overline{X_1}} \cup \overline{\overline{X_2}},\ E_3)} \tag{4.72}$$

Comment: (Rules 4.71 and 4.72) Sequential composition and local declarations are translated in the same manner; the second declarations forms the scope of the first declaration in the translation.

$$\frac{\mathit{vid} \leadsto \mathit{con} \qquad \langle \Gamma(\mathit{con}) = \tau \rightarrow t\_\mathit{exn} \rangle}{\mathcal{S},\ E \vdash \texttt{exception}\ \mathit{vid}\ \langle \texttt{of}\ \mathit{ty} \rangle (:\Gamma) \leadsto (\emptyset,\ \{\textbf{exception}\ (\mathit{con}\ \langle,\ \tau\rangle)\},\ E)} \tag{4.73}$$

$$\frac{\mathcal{S} \vdash \mathit{datbind} \leadsto \overline{\overline{D}}}{\mathcal{S},\ E \vdash \texttt{datatype}\ \mathit{datbind} \leadsto (\overline{\overline{D}},\ \emptyset,\ E)} \tag{4.74}$$

Datatype Bindings $\mathcal{S} \vdash \mathit{datbind} \leadsto \overline{\overline{D}}$

$$\frac{\langle \mathcal{S} \vdash \mathit{datbind} \leadsto \overline{\overline{D}} \rangle \qquad \mathcal{S} \vdash \mathit{conbind} \leadsto \overline{\overline{C}} \qquad \mathit{tycon} \leadsto \mathit{tn}}{\mathcal{S} \vdash \texttt{()}\ \mathit{tycon}\ \texttt{=}\ \mathit{conbind}\ \langle \texttt{and}\ \mathit{datbind} \rangle \leadsto \{\textbf{datatype}\ \mathit{tn}\ \textbf{of}\ \overline{\overline{C}}\}\ \langle \cup\ \overline{\overline{D}} \rangle} \tag{4.75}$$

$$\frac{\begin{array}{l}\langle \mathcal{S} \vdash \mathit{datbind} \leadsto \overline{\overline{D}} \rangle \qquad \mathcal{S} \vdash \mathit{conbind} \leadsto \overline{\overline{C}} \qquad \mathit{tycon} \leadsto \mathit{tn} \\ \mathit{tyvar}_1 \leadsto \alpha^1\ \cdots\ \mathit{tyvar}_n \leadsto \alpha^n\end{array}}{\begin{array}{l}\mathcal{S} \vdash \texttt{(}\mathit{tyvar}_1\texttt{,}\ \cdots\texttt{,}\ \mathit{tyvar}_n\texttt{)}\ \mathit{tycon}\ \texttt{=}\ \mathit{conbind}\ \langle \texttt{and}\ \mathit{datbind} \rangle \leadsto \\ \quad \{\textbf{datatype}\ (\overline{\alpha}^n,\ \mathit{tn})\ \textbf{of}\ \overline{\overline{C}}\}\ \langle \cup\ \overline{\overline{D}} \rangle\end{array}} \tag{4.76}$$

Constructor Bindings $\mathcal{S} \vdash \mathit{conbind} \leadsto \overline{\overline{C}}$

$$\frac{\mathit{vid} \leadsto \mathit{con} \qquad \langle\langle \mathcal{S} \vdash \mathit{conbind} \leadsto \overline{\overline{C}} \rangle\rangle}{\mathcal{S} \vdash \mathit{vid}\ \langle \texttt{of}\ \mathit{ty} \rangle\ \langle\langle \texttt{|}\ \mathit{conbind} \rangle\rangle\ (:\mathit{CE}) \leadsto \{\ \mathit{contype}(\mathit{con},\ \mathit{CE}(\mathit{con})\ \}\ \langle\langle \cup\ \overline{\overline{C}} \rangle\rangle} \tag{4.77}$$

Comment: (Rule:4.77) A Dynamic ML constructor binding, or sequence of constructor bindings, is translated into a set of $\mathcal{M}\Lambda$ constructor declarations $\overline{\overline{C}}$. The type of a constructor is determined from the annotated constructor environment *CE* using the *contype* function defined below (4.78).

$$\begin{array}{lcl} \mathit{contype}(\mathit{con},\ \mathit{tn}) & = & \mathit{con} \\ \mathit{contype}(\mathit{con},\ \tau \rightarrow \mathit{tn}) & = & (\mathit{con},\ \tau) \\ \mathit{contype}(\mathit{con},\ \forall\,\overline{\alpha}^k.\ \mathit{tn}(\overline{\alpha}^k)) & = & \mathit{con} \\ \mathit{contype}(\mathit{con},\ \forall\,\overline{\alpha}^k.\ \tau \rightarrow \mathit{tn}(\overline{\alpha}^k)) & = & (\mathit{con},\ \tau) \end{array} \tag{4.78}$$

Expressions $\mathcal{S} \vdash expr \rightsquigarrow (\overline{\overline{D}},\ \overline{\overline{X}},\ E)$

A Dynamic ML expression *expr* is directly translated into a $\mathcal{M}\Lambda$ expression by the rule $\mathcal{S} \vdash expr \rightsquigarrow (\overline{\overline{D}},\ \overline{\overline{X}},\ E)$ defined below. A Dynamic ML expression may contain datatype and exception declarations, while in $\mathcal{M}\Lambda$ all datatype and exception declarations must appear separately. Therefore, the translated datatypes $\overline{\overline{D}}$ and exceptions $\overline{\overline{X}}$ are returned separately in this rule.

$$\frac{}{\mathcal{S} \vdash scon \rightsquigarrow (\emptyset,\ \emptyset,\ \mathbf{scon}\ scon)} \tag{4.79}$$

$$\frac{vid \rightsquigarrow x \qquad ftvs(\tau) = \emptyset}{\mathcal{S} \vdash vid\ (:\tau) \rightsquigarrow (\emptyset,\ \emptyset,\ \mathbf{var}\ x)} \tag{4.80}$$

$$\frac{vid \rightsquigarrow x \qquad ftvs(\tau) = \{\alpha_1,\ \ldots,\ \alpha_k\}}{\mathcal{S} \vdash vid\ (:\tau) \rightsquigarrow (\emptyset,\ \emptyset,\ \mathbf{var}\ (x,\ (\mathcal{S}(\alpha_1),\ \ldots,\ \mathcal{S}(\alpha_k))))} \tag{4.81}$$

Comment: (Rules 4.80 and 4.81) The type τ is obtained from the annotated syntax tree. If τ has no free type variables, then a monomorphic **var** expression is generated. However, if τ has free type variables, an instance is created using $\mathcal{S}$ to obtain the types for the type variables.

$$\frac{vid \rightsquigarrow con \qquad ftvs(\tau) = \emptyset}{\mathcal{S} \vdash vid\ (:\tau) \rightsquigarrow (\emptyset,\ \emptyset,\ \mathbf{con}\ con)} \tag{4.82}$$

$$\frac{vid \rightsquigarrow con \qquad ftvs(\tau) = \{\alpha_1,\ \ldots,\ \alpha_k\}}{\mathcal{S} \vdash vid\ (:\tau) \rightsquigarrow (\emptyset,\ \emptyset,\ \mathbf{con}\ (con,\ (\mathcal{S}(\alpha_1),\ \ldots,\ \mathcal{S}(\alpha_k))))} \tag{4.83}$$

Comment: (Rules 4.82 and 4.83) Monomorphic and polymorphic constructors are translated in the same manner as variables.

$$\frac{}{\mathcal{S} \vdash \texttt{\{\}} \rightsquigarrow (\emptyset,\ \emptyset,\ \mathbf{scon}\ unit)} \tag{4.84}$$

$$\frac{\begin{array}{l}\mathcal{S} \vdash expr_1 \rightsquigarrow (\overline{\overline{D_1}},\ \overline{\overline{X_1}},\ E_1) \quad \cdots \quad \mathcal{S} \vdash expr_n \rightsquigarrow (\overline{\overline{D_n}},\ \overline{\overline{X_n}},\ E_n) \\ \delta \text{ a permutation of } 1 \ldots n, \text{ where } lab_{\delta(1)} < \cdots < lab_{\delta(n)}\end{array}}{\begin{array}{l}\mathcal{S} \vdash \texttt{\{}lab_1 \texttt{ = } expr_1\texttt{,}\ \cdots\texttt{,}\ lab_n \texttt{ = } expr_n\texttt{\}} \rightsquigarrow \\ \quad (\overline{\overline{D_1}} \cup \cdots \cup \overline{\overline{D_n}},\ \overline{\overline{X_1}} \cup \cdots \cup \overline{\overline{X_n}},\ \mathbf{tuple}\ (E_{\delta(1)},\ \ldots,\ E_{\delta(n)}))\end{array}} \tag{4.85}$$

Comment: (Rules 4.84 and 4.85) A non-empty record is translated into an ordered **tuple**. However, an empty record is translated into the special unit value.

$$\frac{\mathcal{S} \vdash \mathit{expr} \leadsto (\overline{\overline{D_1}},\ \overline{\overline{X_1}},\ E_1) \qquad \mathcal{S},\ E_1 \vdash \mathit{dec} \leadsto (\overline{\overline{D_2}},\ \overline{\overline{X_2}},\ E_2)}{\mathcal{S} \vdash \texttt{let}\ \mathit{dec}\ \texttt{in}\ \mathit{expr}\ \texttt{end} \leadsto (\overline{\overline{D_1}} \cup \overline{\overline{D_2}},\ \overline{\overline{X_1}} \cup \overline{\overline{X_2}},\ E_2)} \tag{4.86}$$

Comment: (Rule 4.86) The expression *expr* is translated before the declaration *dec*, although they will be evaluated in the correct order.

$$\frac{\mathcal{S} \vdash \mathit{expr}_1 \leadsto (\overline{\overline{D_1}},\ \overline{\overline{X_1}},\ E_1) \qquad \mathcal{S} \vdash \mathit{expr}_2 \leadsto (\overline{\overline{D_2}},\ \overline{\overline{X_2}},\ E_2)}{\mathcal{S} \vdash \mathit{expr}_1\ \mathit{expr}_2 \leadsto (\overline{\overline{D_1}} \cup \overline{\overline{D_2}},\ \overline{\overline{X_1}} \cup \overline{\overline{X_2}},\ \mathbf{app}\ (E_1,\ E_2))} \tag{4.87}$$

$$\frac{\mathcal{S} \vdash \mathit{expr} \leadsto (\overline{\overline{D}},\ \overline{\overline{X}},\ E)}{\mathcal{S} \vdash \mathit{expr}\ \texttt{:}\ \mathit{ty} \leadsto (\overline{\overline{D}},\ \overline{\overline{X}},\ E)} \tag{4.88}$$

Comment: (Rule 4.88) The type annotation *ty* is only relevant to elaboration and is therefore ignored.

$$\frac{\begin{array}{l}\mathcal{S} \vdash \mathit{expr} \leadsto (\overline{\overline{D_1}},\ \overline{\overline{X_1}},\ E_1) \qquad x \text{ fresh} \\ \mathcal{S},\ \mathbf{var}\ x,\ \mathbf{raise}\ (\mathbf{var}\ x,\ \mathit{subst}(\mathcal{S},\ \tau)) \vdash \mathit{match} \leadsto (\overline{\overline{D_2}},\ \overline{\overline{X_2}},\ E_2)\end{array}}{\begin{array}{l}\mathcal{S} \vdash \mathit{expr}\ \texttt{handle}\ \mathit{match}\ (:\tau) \leadsto \\ \quad (\overline{\overline{D_1}} \cup \overline{\overline{D_2}},\ \overline{\overline{X_1}} \cup \overline{\overline{X_2}},\ \mathbf{handle}\ E_1\ \mathbf{with}\ (\mathbf{fn}\ (x,\ \text{t\_exn} \to \mathit{subst}(\mathcal{S},\ \tau))\ =\ E_2))\end{array}} \tag{4.89}$$

Comment: (Rule 4.89) An exception handler is translated into a function, taking an exception packet (of type t_exn) as arguments. A fresh variable x is introduced to hold the exception packet. If the handler does not match the exception packet, then the exception is raised again.

$$\frac{\mathcal{S} \vdash \mathit{expr} \leadsto (\overline{\overline{D}},\ \overline{\overline{X}},\ E)}{\mathcal{S} \vdash \texttt{raise}\ \mathit{expr}\ (:\tau) \leadsto (\overline{\overline{D}},\ \overline{\overline{X}},\ \mathbf{raise}\ (E,\ \mathit{subst}(\mathcal{S},\ \tau)))} \tag{4.90}$$

Comment: (Rule 4.90) The type of the raise expression is obtained from the annotated syntax tree. For reasons discussed in chapter 3, the type t_exn is not the type of the **raise**.

$$\frac{x \text{ fresh} \qquad \mathcal{S},\ \mathbf{var}\ x,\ \mathbf{raise}\ (\text{e\_match},\ \mathit{subst}(\mathcal{S},\ \tau)) \vdash \mathit{match} \leadsto (\overline{\overline{D}},\ \overline{\overline{X}},\ E)}{\mathcal{S} \vdash \texttt{fn}\ \mathit{match}\ (:\tau) \leadsto (\overline{\overline{D}},\ \overline{\overline{X}},\ \mathbf{fn}\ (x,\ \mathit{subst}(\mathcal{S},\ \tau))\ =\ E)} \tag{4.91}$$

Comment: (Rule 4.91) The fresh variable x will be bound to the function argument upon application. If this argument does not match any of the patterns in *match*, the exception e_match will be raised.

4.4.1 Pattern Matching

Pattern matching is an integral part of Dynamic ML as it allows functions over datatypes to be expressed in an intuitive manner. The simplest kind of pattern matching is *string-matching*. A string matcher takes two inputs: a pattern string and an object string. The matcher finds the first occurrence (if any) of the pattern in the object string. There are many efficient algorithms for performing string matching, e.g. Knuth-Morris-Pratt. Dynamic ML has a more advanced form of pattern matching, called *term-matching*. This takes two inputs: a *match*, which is a sequence of patterns, and an object *term*. Patterns are incomplete terms with variables. The matcher finds the first pattern (if any) which is matched by the object term. Any variables in the matching pattern are bound to the corresponding values of the term.

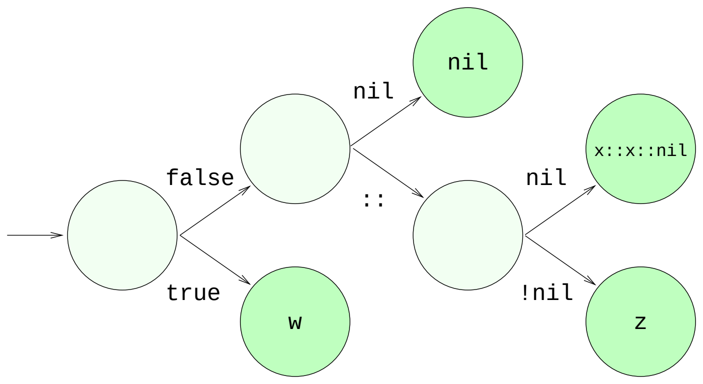
Dynamic ML pattern matching allows complex nested patterns to be expressed, and permits partial overlap between matches. In order to reduce this complexity, a typical Dynamic ML compiler will perform a *match-compilation* step. The match compiler translates complex patterns into simple conditional statements with no nesting or overlap. The match compiler will also warn of non-exhaustive or redundant cases. There are many different algorithms for performing match compilation, e.g. [Pey87], [Ses96], and [Pet99]. However, they are all based on a similar idea. Patterns can be viewed as regular expressions consisting of constructor names, atomic values, and wild-cards. These regular expressions can be easily mapped to tree-based deterministic finite automata. The leaves of the tree correspond to right-hand side expressions, and the internal nodes correspond to conditional tests. This provides an efficient encoding of the patterns without requiring backtracking. Further improvements can be made by eliminating unnecessary arcs, and combining identical states in the automaton. An example match compilation is shown in Figure 4.6.

In this chapter, our interest in match compilation is not primarily motivated by efficiency, rather the fact that $\mathcal{M}\Lambda$ does not contain pattern matching. Therefore, we must use a technique similar to the one outlined above to convert Dynamic ML pattern matching into simpler $\mathcal{M}\Lambda$ conditionals. In order to present the translation is a clean manner, we will adopt the most naive approach to pattern matching; each pattern is translated into a sequence of conditional tests which are applied in turn until a match is found. This is somewhat wasteful, since many of the tests may be repeated. However, we note that an actual implementation of the translation would benefit from a more efficient approach, such as the one outlined above.

1. Standard ML Function:

```
fun test (false, nil) = nil
  | test (true, w) = w
  | test (false, x :: nil) = x :: x :: nil
  | test (false, y :: z) = z;
```

2. Deterministic Finite Automata:



3. Optimised Function:

```
fun test (a, b) =
  case a of true => b
         |  false =>
             case b of nil => nil
                    |  (c :: d) =>
                         case d of nil => c :: c :: nil
                                |  _  => d;
```

Figure 4.6: *Match Compilation.*

The translation of Dynamic ML pattern matching into $\mathcal{M}\Lambda$ expressions is defined by a set of rules in the following section. It is important to remember that we are performing a translation operation, rather than an evaluation. In some instances, the expressions are translated in a different order from the one which will be used for evaluation. In particular, the patterns are translated in reverse order, starting with the last pattern and working backwards to the first pattern. However, when the resulting expressions are composed at the end of the translation, the outcome will be a single expression which will evaluate in the correct left-right order.

Match Compilation $\mathcal{S},\ E_1,\ E_2 \vdash \mathit{match} \leadsto (\overline{\overline{D}},\ \overline{\overline{X}},\ E_3)$

The translation of a Dynamic ML *match* expression is defined below using a pair of rules of the form $\mathcal{S},\ E_1,\ E_2 \vdash \mathit{match} \leadsto E_3$. The $\mathcal{M}\Lambda$ expression E_1 is the translated object term against which the patterns will be matched. The expression E_2 is evaluated if a particular pattern is not matched. As the patterns are processed in reverse order, this expression will initially correspond to the failure of pattern matching, i.e. a `Match` exception, unless the *match* is itself part of an exception handler (Rule 4.89). Following this, the expression E_2 will accumulate the patterns that have already been translated. The result of the translation is a expression E_3 which corresponds to an evaluation where each of the patterns are applied in turn. The translation is expressed recursively. Note that Rule 4.93 immediately invokes itself recursively on the remaining *match* in the sequence. Hence, the base case (Rule 4.92) is applied to the last *match* in the sequence.

$$\frac{\mathcal{S} \vdash \mathit{expr} \leadsto (\overline{\overline{D}},\ \overline{\overline{X}},\ E_3) \qquad \mathcal{S},\ E_1,\ E_3,\ E_2 \vdash \mathit{pat} \leadsto E_4}{\mathcal{S},\ E_1,\ E_2 \vdash \mathit{pat}\ \texttt{=>}\ \mathit{expr} \leadsto (\overline{\overline{D}},\ \overline{\overline{X}},\ E_4)} \tag{4.92}$$

$$\frac{\mathcal{S},\ E_1,\ E_2 \vdash \mathit{match} \leadsto (\overline{\overline{D_1}},\ \overline{\overline{X_1}},\ E_3) \qquad \mathcal{S} \vdash \mathit{expr} \leadsto (\overline{\overline{D_2}},\ \overline{\overline{X_2}},\ E_4) \qquad \mathcal{S},\ E_1,\ E_4,\ E_3 \vdash \mathit{pat} \leadsto E_5}{\mathcal{S},\ E_1,\ E_2 \vdash \mathit{pat}\ \texttt{=>}\ \mathit{expr}\ \texttt{|}\ \mathit{match} \leadsto (\overline{\overline{D_1}} \cup \overline{\overline{D_2}},\ \overline{\overline{X_1}} \cup \overline{\overline{X_2}},\ E_5)} \tag{4.93}$$

Pattern Compilation $\mathcal{S},\ E_1,\ E_2,\ E_3 \vdash \mathit{pat} \leadsto E_4$

Patterns are translated by a series of rules of the form $\mathcal{S},\ E_1,\ E_2,\ E_3 \vdash \mathit{pat} \leadsto E_4$ which we define below. The expression E_1 is the object term against which the pattern *pat* is to be matched. The expression E_2 is to be evaluated if the pattern is matched, and the expression E_3 is to be evaluated otherwise. The expression E_4 is the result of the translation. It is helpful to think of the rules below as a translation of an expression `if` $E_1 = \mathit{pat}$ `then` E_2 `else` E_3 for each different form of *pat*.

$$\frac{}{\mathcal{S},\ E_1,\ E_2,\ E_3 \vdash \texttt{\_} \leadsto E_2} \tag{4.94}$$

Comment: (Rule 4.94) The wild-card pattern _ is always matched, so the expression E_2 is simply returned. Note that the expression E_3, i.e. any pattern which follows the wild-card, is ignored.

$$\frac{}{\begin{array}{l}\mathcal{S},\ E_1,\ E_2,\ E_3 \vdash \mathit{scon} \rightsquigarrow \\ \quad \textbf{switch}\ (\textbf{app}\ (\mathrm{EQ}_{tn_{scon}},\ \textbf{tuple}\ (E_1,\ \textbf{scon}\ \mathit{scon}))) \\ \qquad \textbf{case}\ (\text{c\_true}\ \mapsto\ E_2,\ E_3)\end{array}} \tag{4.95}$$

Comment: (Rule 4.95) We compare the special constant *scon* with E_1 using the EQ function from the Basis.

$$\frac{\mathit{vid} \rightsquigarrow x}{\mathcal{S},\ E_1,\ E_2,\ E_3 \vdash \mathit{vid}\ (:\tau) \rightsquigarrow \textbf{let}\ (x,\ \mathit{subst}(\mathcal{S},\ \tau))\ =\ E_1\ \textbf{in}\ E_2} \tag{4.96}$$

Comment: (Rule 4.96) A variable which occurs in a pattern is bound to E_1 in the scope of E_2. Note that the type of the variable is obtained from the annotated type.

$$\frac{\mathit{vid} \rightsquigarrow \mathit{con}}{\mathcal{S},\ E_1,\ E_2,\ E_3 \vdash \mathit{vid} \rightsquigarrow \textbf{switch}\ E_1\ \textbf{case}\ (\mathit{con}\ \mapsto\ E_2,\ E_3)} \tag{4.97}$$

Comment: (Rule 4.97) A constructor pattern is translated directly into a **switch** expression.

$$\frac{}{\begin{array}{l}\mathcal{S},\ E_1,\ E_2,\ E_3 \vdash \texttt{\{\}} \rightsquigarrow \\ \quad \textbf{switch}\ (\textbf{app}\ (\mathrm{EQ}_{\text{t\_unit}},\ \textbf{tuple}\ (E_1,\ \textbf{scon}\ \mathit{unit}))) \\ \qquad \textbf{case}\ (\text{c\_true}\ \mapsto\ E_2,\ E_3)\end{array}} \tag{4.98}$$

Comment: (Rule 4.98) An empty record pattern, representing the unit value, is compared using the basis function $\mathrm{EQ}_{\text{t\_unit}}$.

$$\frac{\begin{array}{l}\delta \text{ a permutation of } 1 \ldots n, \text{ where } \mathit{lab}_{\delta(1)} < \cdots < \mathit{lab}_{\delta(n)} \\ \mathcal{S},\ \textbf{select}\ (n,\ E_1),\ E_2,\ E_3 \vdash \mathit{pat}_{\sigma(n)} \rightsquigarrow E_4^1 \quad \cdots \\ \quad \mathcal{S},\ \textbf{select}\ (1,\ E_1),\ E_4^{n-1},\ E_3 \vdash \mathit{pat}_{\sigma(1)} \rightsquigarrow E_4^n\end{array}}{\mathcal{S},\ E_1,\ E_2,\ E_3 \vdash \texttt{\{}\mathit{lab}_1 \texttt{=} \mathit{pat}_1\texttt{,}\ \cdots\texttt{,}\ \mathit{lab}_n \texttt{=} \mathit{pat}_n\texttt{\}} \rightsquigarrow E_4^n} \tag{4.99}$$

Comment: (Rule 4.99) A record pattern is processed in translated in reverse order, as defined by the permutation function δ. A **select** expression is used as E_1 to obtain the correct component of the record.

$$\frac{\mathcal{S},\ E_1,\ E_2,\ E_3 \vdash \mathit{pat} \rightsquigarrow E_4}{\mathcal{S},\ E_1,\ E_2,\ E_3 \vdash \mathit{pat} : \mathit{ty} \rightsquigarrow E_4} \tag{4.100}$$

Comment: (Rule 4.100) The type constraint ty is only relevant to the elaboration phase and is simply ignored in the translation.

$$\frac{\begin{array}{l} \mathit{vid} \rightsquigarrow \mathit{con} \qquad \mathit{ftvs}(\tau) = \emptyset \\ \mathcal{S},\ \mathbf{decon}(\mathit{con},\ E_1),\ E_2,\ E_3 \vdash \mathit{pat} \rightsquigarrow E_4 \end{array}}{\mathcal{S},\ E_1,\ E_2,\ E_3 \vdash \mathit{vid}\,\mathit{pat}\ (: \tau) \rightsquigarrow E_4} \tag{4.101}$$

$$\frac{\begin{array}{l} \mathit{vid} \rightsquigarrow \mathit{con} \qquad \mathit{ftvs}(\tau) = \{\alpha_1,\ \ldots,\ \alpha_k\} \\ \mathcal{S},\ \mathbf{decon}(\mathit{con},\ (\mathcal{S}(\alpha_1),\ \ldots,\ \mathcal{S}(\alpha_k)),\ E_1),\ E_2,\ E_3 \vdash \mathit{pat} \rightsquigarrow E_4 \end{array}}{\mathcal{S},\ E_1,\ E_2,\ E_3 \vdash \mathit{vid}\,\mathit{pat}\ (: \tau) \rightsquigarrow E_4} \tag{4.102}$$

Comment: (Rules 4.101 and 4.102) The match a constructed pattern, we must deconstruct E_1.

$$\frac{\mathcal{S},\ E_1,\ E_2,\ E_3 \vdash \mathit{pat} \rightsquigarrow E_4 \qquad \mathit{vid} \rightsquigarrow x}{\mathcal{S},\ E_1,\ E_2,\ E_3 \vdash \mathit{vid}\ \langle : \mathit{ty}\rangle\ \texttt{as}\ \mathit{pat}\ (: \tau) \rightsquigarrow \mathbf{let}\ (x,\ \mathit{subst}(\mathcal{S},\ \tau))\ =\ E_1\ \mathbf{in}\ E_4} \tag{4.103}$$

Comment: (Rule 4.103) A layered pattern is translated into a **let** expression such that the variable x is only visible to E_4.

4.5 Example Translation

We now present a detailed example to illustrate the translation process in operation. Our example performs the translation of the factorial function from Dynamic ML to $\mathcal{M}\Lambda$. Figure 4.7 illustrates the difference between the Standard ML and Dynamic ML definitions. The Dynamic ML form corresponds to the removal of all of the derived forms given in the Definition of Standard ML [MTHM97], and provides the starting point for our translation.

The first step in the translation is the elaboration phase, illustrated in Figure 4.8. We have omitted the substitutions $\mathcal{S}$ for clarity. The figure shows the syntax tree for the program, annotated with type inference information. The syntax of the program appears in boxes to separate it from the types. The inference operation begins at the bottom-left of this diagram and proceeds in an approximately clockwise manner. The inference rules invoked at each step are shown in parentheses. The result is the initial environment Γ extended with the mapping $fact \mapsto$ t_int $\rightarrow$ t_int. The second and final step is the generation of $\mathcal{M}\Lambda$ code, illustrated in Figure 4.9. We have omitted the datatype and exception parts of the program, since neither are required in the definition of the factorial function. As noted earlier in this chapter, the elaboration and generation phases perform a similar traversal of the syntax tree, and could therefore be combined for efficiency in an implementation.

The $\mathcal{M}\Lambda$ factorial program, resulting from our translation operation, is shown in Figure 4.7. Comparing this program to the hand-translated version in Chapter 2 reveals that there is some redundancy: the **let** expression could be removed and the variable x substituted in place of n. As stated earlier, our translator does not currently perform any optimisation on the code, and pattern matching is only translated using a naive algorithm. The code could be significantly improved by performing a post-processing optimisation stage with a number of λ-calculus based optimisations, though these techniques are well known and we do not present further details here.

Standard ML:

```
fun fact 0 = 1
  | fact n = n * fact(n - 1);
```

Dynamic ML:

```
val () rec {lab1 = fact} =
           {lab1 = fn 0 => 1 |
                       n => * {1 = n, 2 = fact (- {1 = n, 2 = 1})}
```

$\mathcal{M}\Lambda$ Translation:

```
(∅, ∅,
  fix ((fact, t_int → t_int) =
    (fn (x, t_int → t_int) =
      switch (app (=, tuple (var x, scon 0)))
      case (c_true ↦ scon 1,
            let (n, t_int) = var x
            in app (var ∗, tuple (var n,
              app(var fact, app (var −, tuple (var n, scon 1)))))))
  in tuple (var fact))
```

Figure 4.7: *Factorial Translation Example.*

$$\frac{\texttt{n} \rightsquigarrow n \quad \Gamma''(n) = \alpha_2}{\Gamma \vdash \boxed{\texttt{n}} : \alpha_2}\,(4.30)$$

$$\frac{\texttt{-} \rightsquigarrow - \quad \Gamma''(-) = (\text{t\_int},\ \text{t\_int}) \to \text{t\_int}}{\Gamma \vdash \boxed{\texttt{-}} : (\text{t\_int},\ \text{t\_int}) \to \text{t\_int}}\,(4.30)$$

$$\frac{\Gamma \vdash \boxed{\texttt{n}} : \alpha_2 \qquad \Gamma \vdash \boxed{\texttt{1}} :_{(4.43)} \text{t\_int}}{\Gamma'' \vdash \boxed{\texttt{\{1 = n, 2 = 1\}}} : (\alpha_2,\ \text{t\_int})}\,(4.33)$$

$$\frac{\texttt{fact} \rightsquigarrow fact \quad \Gamma''(fact) = \alpha_1}{\Gamma'' \vdash \boxed{\texttt{fact}} : \alpha_1}\,(4.30)$$

$$\frac{\Gamma \vdash \boxed{\texttt{-}} : (\text{t\_int},\ \text{t\_int}) \to \text{t\_int} \quad \Gamma'' \vdash \boxed{\texttt{\{1 = n, 2 = 1\}}} : (\alpha_2,\ \text{t\_int}) \quad \alpha_3 \text{ fresh} \quad unify((\text{t\_int},\ \text{t\_int}) \to \text{t\_int},\ (\alpha_2,\ \text{t\_int}) \to \alpha_3}{\Gamma'' \vdash \boxed{\texttt{- \{1 = n, 2 = 1\}}} : \text{t\_int}}\,(4.35)$$

$$\frac{\texttt{n} \rightsquigarrow n \quad \Gamma''(n) = \alpha_2}{\Gamma'' \vdash \boxed{\texttt{n}} : \alpha_2}\,(4.30)$$

$$\frac{\Gamma'' \vdash \boxed{\texttt{fact}} : \alpha_1 \quad \Gamma'' \vdash \boxed{\texttt{- \{1 = n, 2 = 1\}}} : \text{t\_int} \quad \alpha_4 \text{ fresh} \quad unify(\alpha_1,\ \text{t\_int} \to \alpha_4)}{\Gamma'' \vdash \boxed{\texttt{fact (- \{1 = n, 2 = 1\})}} : \text{t\_int}}\,(4.35)$$

$$\frac{\texttt{*} \rightsquigarrow * \quad \Gamma''(*) = (\text{t\_int},\ \text{t\_int}) \to \text{t\_int}}{\Gamma'' \vdash \boxed{\texttt{*}} : (\text{t\_int},\ \text{t\_int}) \to \text{t\_int}}\,(4.30)$$

$$\frac{\Gamma'' \vdash \boxed{\texttt{n}} : \alpha_2 \quad \Gamma'' \vdash \boxed{\texttt{fact (- \{1 = n, 2 = 1\})}} : \text{t\_int}}{\Gamma \vdash \boxed{\texttt{\{1 = n, 2 = fact (- \{1 = n, 2 = 1\})\}}} : (\alpha_2,\ \text{t\_int} \to \alpha_4)}\,(4.33)$$

$$\frac{\texttt{n} \rightsquigarrow n \quad \alpha_2 \text{ fresh}}{\Gamma' \vdash \boxed{\texttt{n}} : (\alpha_2,\ \{n\ \mapsto\ \alpha_2\})}\,(4.44)$$

$$\frac{\Gamma'' \vdash \boxed{\texttt{*}} : (\text{t\_int},\ \text{t\_int}) \to \text{t\_int} \quad \Gamma \vdash \boxed{\texttt{\{1 = n, 2 = fact (- \{1 = n, 2 = 1\})\}}} : (\alpha_2,\ \text{t\_int} \to \alpha_4) \quad \alpha_5 \text{ fresh} \quad unify((\text{t\_int},\ \text{t\_int}) \to \text{t\_int},\ (\alpha_2,\ \text{t\_int}) \to \alpha_5)}{\Gamma'[n\ \mapsto\ \alpha_2] \vdash \boxed{\texttt{* \{1 = n, 2 = fact (- \{1 = n, 2 = 1\})\}}} : \text{t\_int}}\,(4.35)$$

$$\frac{\Gamma' \vdash \boxed{\texttt{n}} : (\alpha_2,\ \{n\ \mapsto\ \alpha_2\}) \qquad \Gamma'[n\ \mapsto\ \alpha_2] \vdash \boxed{\texttt{* \{1 = n, 2 = fact (- \{1 = n, 2 = 1\})\}}} : \text{t\_int}}{\Gamma' \vdash \boxed{\texttt{n => * \{1 = n, 2 = fact (- \{1 = n, 2 = 1\})\}}} : \alpha_2 \to \text{t\_int}}\,(4.40)$$

$$\frac{\Gamma' \vdash \boxed{\texttt{0}} :_{(4.43)} \text{t\_int} \quad \Gamma' \vdash \boxed{\texttt{1}} :_{(4.29)} \text{t\_int} \quad \Gamma' \vdash \boxed{\texttt{n => * \{1 = n, 2 = fact (- \{1 = n, 2 = 1\})\}}} : \alpha_2 \to \text{t\_int} \quad unify(\alpha_2 \to \text{t\_int},\ \text{t\_int} \to \text{t\_int})}{\Gamma' \vdash \boxed{\texttt{0 => 1 | n => * \{1 = n, 2 = fact (- \{1 = n, 2 = 1\})\}}} : \text{t\_int} \to \text{t\_int}}\,(4.41)$$

$$\frac{\texttt{fact} \rightsquigarrow fact \quad \alpha_1 \text{ fresh}}{\Gamma \vdash \boxed{\texttt{fact}} : (\alpha_1,\ \{fact\ \mapsto\ \alpha_1\})}\,(4.30)$$

$$\frac{\Gamma' \vdash \boxed{\texttt{0 => 1 | n => * \{1 = n, 2 = fact (- \{1 = n, 2 = 1\})\}}} : \text{t\_int} \to \text{t\_int}}{\Gamma' \vdash \boxed{\texttt{fn 0 => 1 | n => * \{1 = n, 2 = fact (- \{1 = n, 2 = 1\})\}}} : (\text{t\_int} \to \text{t\_int})}\,(4.39)$$

$$\frac{\Gamma \vdash \boxed{\texttt{fact}} : (\alpha_1,\ \{fact\ \mapsto\ \alpha_1\})}{\Gamma \vdash \boxed{\texttt{\{lab1 = fact\}}} : ((\alpha_1),\ \{fact\ \mapsto\ \alpha_1\})}\,(4.33)$$

$$\frac{\Gamma' \vdash \boxed{\texttt{fn 0 => 1 | n => * \{1 = n, 2 = fact (- \{1 = n, 2 = 1\})\}}} : (\text{t\_int} \to \text{t\_int})}{\Gamma[fact\ \mapsto\ \alpha_1] \vdash \boxed{\texttt{\{lab1 = fn 0 => 1 | n => * \{1 = n, 2 = fact (- \{1 = n, 2 = 1\})\}}} : (\text{t\_int} \to \text{t\_int})}\,(4.33)$$

$$\frac{\Gamma \vdash \boxed{\texttt{\{lab1 = fact\}}} : ((\alpha_1),\ \{fact\ \mapsto\ \alpha_1\}) \quad \Gamma[fact\ \mapsto\ \alpha_1] \vdash \boxed{\texttt{\{lab1 = fn 0 => 1 | n => * \{1 = n, 2 = fact (- \{1 = n, 2 = 1\})\}}} : (\text{t\_int} \to \text{t\_int}) \quad unify((\alpha_1),\ (\text{t\_int} \to \text{t\_int}))}{\Gamma \vdash \boxed{\texttt{val () rec \{lab1 = fact\} = \{lab1 = fn 0 => 1 | n => * \{1 = n, 2 = fact (- \{1 = n, 2 = 1\})\}}} : \Gamma[fact\ \mapsto\ \text{t\_int} \to \text{t\_int}]}\,(4.19)$$

$$\frac{\Gamma \vdash \boxed{\texttt{val () rec \{lab1 = fact\} = \{lab1 = fn 0 => 1 | n => * \{1 = n, 2 = fact (- \{1 = n, 2 = 1\})\}}} : \Gamma[fact\ \mapsto\ \text{t\_int} \to \text{t\_int}]}{\Gamma \vdash \boxed{\texttt{val () rec \{lab1 = fact\} = \{lab1 = fn 0 => 1 | n => * \{1 = n, 2 = fact (- \{1 = n, 2 = 1\})\}}} : \Gamma[fact\ \mapsto\ \text{t\_int} \to \text{t\_int}]}\,(4.16)$$

Figure 4.8: *Type Inference Phase.*

$$
\dfrac{
\begin{array}{l}
\dfrac{
\begin{array}{l}
\dfrac{
\dfrac{\texttt{*} \rightsquigarrow *}{\vdash \boxed{\texttt{*}} \rightsquigarrow \mathbf{var}\ *}\,(4.80)
\quad
\dfrac{
\dfrac{\texttt{n} \rightsquigarrow n}{\vdash \boxed{\texttt{n}} \rightsquigarrow \mathbf{var}\ n}\,(4.80)
\quad
\dfrac{
\vdash \boxed{\texttt{fact}} \rightsquigarrow \mathbf{var}\ \mathit{fact}
\quad
\dfrac{
\dfrac{\texttt{-} \rightsquigarrow -}{\vdash \boxed{\texttt{-}} \rightsquigarrow \mathbf{var}\ -}\,(4.80)
\quad
\dfrac{
\dfrac{\texttt{n} \rightsquigarrow n}{\vdash \boxed{\texttt{n}} \rightsquigarrow \mathbf{var}\ n}\,(4.80)
\quad
\vdash \boxed{\texttt{1}} \rightsquigarrow_{(4.79)} \mathbf{scon}\ 1
}{\vdash \boxed{\texttt{\{1 = n, 2 = 1\}}} \rightsquigarrow \mathbf{tuple}(\mathbf{var}\ n, \mathbf{scon}\ 1)}\,(4.85)
}{\vdash \boxed{\texttt{- \{1 = n, 2 = 1\}}} \rightsquigarrow \mathbf{app}(\mathbf{var}\ -,\ \mathbf{tuple}\ (\mathbf{var}\ n, \mathbf{scon}\ 1))}\,(4.87)
}{\vdash \boxed{\texttt{fact (- \{1 = n, 2 = 1\})}} \rightsquigarrow \mathbf{app}(\mathbf{var}\ \mathit{fact},\ \mathbf{app}\ (\mathbf{var}\ -,\ \mathbf{tuple}\ (\mathbf{var}\ n, \mathbf{scon}\ 1)))}\,(4.87)
}{\vdash \boxed{\texttt{\{1 = n, 2 = fact (- \{1 = n, 2 = 1\})}} \rightsquigarrow \mathbf{tuple}\ (\mathbf{var}\ n,\ \mathbf{app}(\mathbf{var}\ \mathit{fact},\ \mathbf{app}\ (\mathbf{var}\ -,\ \mathbf{tuple}\ (\mathbf{var}\ n, \mathbf{scon}\ 1))))}\,(4.85)
}{\vdash \boxed{\texttt{* \{1 = n, 2 = fact (- \{1 = n, 2 = 1\})}} \rightsquigarrow \mathbf{app}\ (\mathbf{var}\ *,\ \mathbf{tuple}\ (\mathbf{var}\ n,\ \mathbf{app}(\mathbf{var}\ \mathit{fact},\ \mathbf{app}\ (\mathbf{var}\ -,\ \mathbf{tuple}\ (\mathbf{var}\ n, \mathbf{scon}\ 1)))))}\,(4.87)
\qquad
\dfrac{\texttt{n} \rightsquigarrow n}{\uparrow}\,(4.96)
\\
\mathbf{var}\ x,\ \mathbf{app}\ (\mathbf{var}\ *,\ \mathbf{tuple}\ (\mathbf{var}\ n,\ \mathbf{app}(\mathbf{var}\ \mathit{fact},\ \mathbf{app}\ (\mathbf{var}\ -,\ \mathbf{tuple}\ (\mathbf{var}\ n, \mathbf{scon}\ 1))))),\ \mathbf{raise}\ (\text{e_bind}, \text{t_int}) \vdash \boxed{\texttt{n}} \rightsquigarrow
\\
\quad \mathbf{let}\ (n,\ \text{t_int}) = \mathbf{var}\ x\ \mathbf{in}\ \mathbf{app}\ (\mathbf{var}\ *,\ \mathbf{tuple}\ (\mathbf{var}\ n,\ \mathbf{app}(\mathbf{var}\ \mathit{fact},\ \mathbf{app}\ (\mathbf{var}\ -,\ \mathbf{tuple}\ (\mathbf{var}\ n, \mathbf{scon}\ 1)))))
\end{array}
}{
\begin{array}{l}
\mathbf{var}\ x,\ \mathbf{raise}\ (\text{e_bind}, \text{t_int}) \vdash \boxed{\texttt{n => * \{1 = n, 2 = fact (- \{1 = n, 2 = 1\})\}}} \rightsquigarrow
\\
\quad \mathbf{let}\ (n,\ \text{t_int}) = \mathbf{var}\ x\ \mathbf{in}\ \mathbf{app}\ (\mathbf{var}\ *,\ \mathbf{tuple}\ (\mathbf{var}\ n,\ \mathbf{app}(\mathbf{var}\ \mathit{fact},\ \mathbf{app}\ (\mathbf{var}\ -,\ \mathbf{tuple}\ (\mathbf{var}\ n, \mathbf{scon}\ 1)))))
\end{array}
}\,(4.92)
\\
\vdash \boxed{\texttt{1}} \rightsquigarrow_{(4.79)} \mathbf{scon}\ 1
\qquad
\begin{array}{l}
\mathbf{var}\ x,\ \mathbf{scon}\ 1,\ \mathbf{let}\ (n,\ \text{t_int}) = \mathbf{var}\ x\ \mathbf{in}\ \mathbf{app}\ (\mathbf{var}\ *,\ \mathbf{tuple}\ (\mathbf{var}\ n,\ \mathbf{app}(\mathbf{var}\ \mathit{fact},\ \mathbf{app}\ (\mathbf{var}\ -,\ \mathbf{tuple}\ (\mathbf{var}\ n, \mathbf{scon}\ 1))))) \vdash \boxed{\texttt{0}} \rightsquigarrow_{(4.95)}
\\
\quad \mathbf{switch}\ (\mathbf{app}\ (=,\ \mathbf{tuple}\ (\mathbf{var}\ x,\ \mathbf{scon}\ 0)))
\\
\quad \mathbf{case}\ (\text{c_true}\ \mapsto\ \mathbf{scon}\ 1,\ \mathbf{let}\ (n,\ \text{t_int}) = \mathbf{var}\ x\ \mathbf{in}\ \mathbf{app}\ (\mathbf{var}\ *,\ \mathbf{tuple}\ (\mathbf{var}\ n,\ \mathbf{app}(\mathbf{var}\ \mathit{fact},\ \mathbf{app}\ (\mathbf{var}\ -,\ \mathbf{tuple}\ (\mathbf{var}\ n, \mathbf{scon}\ 1))))))
\end{array}
\end{array}
}{
\begin{array}{l}
\text{fact} \rightsquigarrow \mathit{fact} \qquad x\ \text{fresh} \qquad \mathbf{var}\ x,\ \mathbf{raise}\ (\text{e_bind}, \text{t_int}) \vdash \boxed{\texttt{0 => 1 | n => * \{1 = n, 2 = fact (- \{1 = n, 2 = 1\})\}}} \rightsquigarrow
\\
\qquad\qquad\qquad\qquad \mathbf{switch}\ (\mathbf{app}\ (=,\ \mathbf{tuple}\ (\mathbf{var}\ x,\ \mathbf{scon}\ 0)))
\\
\qquad\qquad\qquad\qquad \mathbf{case}\ (\text{c_true}\ \mapsto\ \mathbf{scon}\ 1,\ \mathbf{let}\ (n,\ \text{t_int}) = \mathbf{var}\ x\ \mathbf{in}\ \mathbf{app}\ (\mathbf{var}\ *,\ \mathbf{tuple}\ (\mathbf{var}\ n,\ \mathbf{app}(\mathbf{var}\ \mathit{fact},\ \mathbf{app}\ (\mathbf{var}\ -,\ \mathbf{tuple}\ (\mathbf{var}\ n, \mathbf{scon}\ 1))))))
\end{array}
}\,(4.93)
$$

$$
\dfrac{
\dfrac{
\begin{array}{l}
\text{fact} \rightsquigarrow \mathit{fact} \qquad x\ \text{fresh} \qquad \mathbf{var}\ x,\ \mathbf{raise}\ (\text{e_bind}, \text{t_int}) \vdash \boxed{\texttt{0 => 1 | n => * \{1 = n, 2 = fact (- \{1 = n, 2 = 1\})\}}} \rightsquigarrow \ \ldots
\end{array}
}{
\begin{array}{l}
VE \in \Gamma = \{\mathit{fact}\ \mapsto\ \text{t_int} \to \text{t_int}\} \qquad \mathbf{tuple}\ (\mathbf{var}\ \mathit{fact}) \vdash \boxed{\texttt{val () rec \{lab1 = fact\} = \{lab1 = fn 0 => 1 | n => * \{1 = n, 2 = fact (- \{1 = n, 2 = 1\})\}}} \rightsquigarrow
\\
\qquad \mathbf{fix}\ ((\mathit{fact},\ \text{t_int} \to \text{t_int}) = (\mathbf{fn}\ (x,\ \text{t_int} \to \text{t_int}) = \mathbf{switch}\ (\mathbf{app}\ (=,\ \mathbf{tuple}\ (\mathbf{var}\ x,\ \mathbf{scon}\ 0)))\ \mathbf{case}\ (\text{c_true}\ \mapsto\ \mathbf{scon}\ 1,
\\
\qquad\quad \mathbf{let}\ (n,\ \text{t_int}) = \mathbf{var}\ x\ \mathbf{in}\ \mathbf{app}\ (\mathbf{var}\ *,\ \mathbf{tuple}\ (\mathbf{var}\ n,\ \mathbf{app}(\mathbf{var}\ \mathit{fact},\ \mathbf{app}\ (\mathbf{var}\ -,\ \mathbf{tuple}\ (\mathbf{var}\ n, \mathbf{scon}\ 1)))))))
\\
\qquad \mathbf{in}\ \mathbf{tuple}\ (\mathbf{var}\ \mathit{fact}))
\end{array}
}\,(4.70)
}{
\begin{array}{l}
\Gamma \vdash \boxed{\texttt{val () rec \{lab1 = fact\} = \{lab1 = fn 0 => 1 | n => * \{1 = n, 2 = fact (- \{1 = n, 2 = 1\})\}}} \rightsquigarrow
\\
\mathbf{fix}\ ((\mathit{fact},\ \text{t_int} \to \text{t_int}) = (\mathbf{fn}\ (x,\ \text{t_int} \to \text{t_int}) = \mathbf{switch}\ (\mathbf{app}\ (=,\ \mathbf{tuple}\ (\mathbf{var}\ x,\ \mathbf{scon}\ 0)))\ \mathbf{case}\ (\text{c_true}\ \mapsto\ \mathbf{scon}\ 1,
\\
\quad \mathbf{let}\ (n,\ \text{t_int}) = \mathbf{var}\ x\ \mathbf{in}\ \mathbf{app}\ (\mathbf{var}\ *,\ \mathbf{tuple}\ (\mathbf{var}\ n,\ \mathbf{app}(\mathbf{var}\ \mathit{fact},\ \mathbf{app}\ (\mathbf{var}\ -,\ \mathbf{tuple}\ (\mathbf{var}\ n, \mathbf{scon}\ 1))))))))\ \mathbf{in}\ \mathbf{tuple}\ (\mathbf{var}\ \mathit{fact}))
\end{array}
}\,(4.68)
$$

Figure 4.9: *Code Generation Phase.*

4.6 Further Work

The definition of the Standard ML language as presented in [MTHM97] contains a number of 'dark alleys' [Kah93], resulting from ambiguities in the semantics, and unexpected combinations of operations. Consequently, these ambiguities are also present in Dynamic ML. We have attempted to provide a translation for the most common cases in this chapter. However, it is probable that there are a number of omissions in the translation when dealing with the less common cases, particularly concerning generativity and equality. Nonetheless, the purpose of this chapter is simply to illustrate that Dynamic ML can be translated into $\mathcal{M}\Lambda$. Ensuring that the translation is correct in all possible cases is left as further work. In later chapters, when dealing purely with the simpler $\mathcal{M}\Lambda$ language, we will be considerably more rigorous in our approach.

It is worth noting that our translation is similar in style to [HS97] which details the translation from Standard ML into a lambda language called IL (Internal Language). IL is based on the translucent-sum module calculi [Lil97] which provides such features as first-order modules and higher-order functors. Datatypes and polymorphism are also translated into module-level operations. Although this formalism is very powerful, we have not adopted it here as we feel that their module system would considerably obfuscate the code-replacement operation which we are seeking to define. In Chapter 7 we discuss a simpler approach, inspired by the ML-Kit compiler [Els99], in which the module system is treated as a compile-time *linking* language, and no actual module code is generated.

Chapter 5

Dynamic Semantics of $\mathcal{M}\Lambda$

A typed programming language is said to obey the *phase distinction* [Car88] if the type of any expression in the language can be determined without evaluating any other expressions. This allows the semantics of the language to be split into a *static semantics* of elaboration and a *dynamic semantics* of evaluation. $\mathcal{M}\Lambda$ is one such language which obeys the phase-distinction. As we have seen in the Chapter 3, our static semantics does not rely on the evaluation of any expressions. We may therefore define the dynamic semantics of the language in this chapter entirely separately from the static semantics. The loss of the phase distinction manifests itself as the inability to perform compilation of the language, hence its retention is an important pragmatic consideration.

The dynamic semantics formalise the execution behaviour of programs. Our definition is based on a series of equivalence relations, which we call *transitions*, between states of an abstract machine. This machine describes the execution and memory allocation behaviour of an implementation of the language, except that it abstracts from the allocation of environments as this simply adds extra baggage to the rules; for a treatment of this topic see [MMH96]. The organisation of the $\mathcal{M}\Lambda$ abstract machine has some features in common with the $\lambda_{\mathrm{gc}}^{\rightarrow\forall}$ abstract machine [MH96] which is used in the formal description of the behaviour of the TIL/ML compiler. However, our relations differ considerably as $\mathcal{M}\Lambda$ does not adopt a continuation-passing style or named-form representations.

The abstract machine is type-preserving in that type information is maintained throughout the evaluation process. As stated in the introduction, this type information will allow us to provide a clean definition of garbage-collection and code-replacement. The presence of type-information in the dynamic semantics does not violate the phase distinction as we are not performing any type checking. We can only evaluate well-typed programs, as defined by the static semantics, and each stage of the evaluation preserves this well-typing.

5.1 Abstract Machine Syntax

We only consider the dynamic semantics of the *sequential* part of $\mathcal{M}\Lambda$ in this chapter, i.e. excluding the threading and communication primitives. The dynamic semantics for the full *distributed* language, presented in Chapter 8, is an extension of the sequential case.

The syntax of our abstract machine is defined in Figure 5.1. The state of the abstract machine is defined by a four-tuple $(H,\ \Delta,\ XS,\ RS)$ of a heap, an environment, an exception stack, and a result stack. We denote a stack as a dotted sequence, e.g. $(a \cdot b \cdot c)$. The leftmost element of the sequence is the top of the stack, and a pair of adjacent parentheses () is used to represent an empty stack.

Machine State	M	::=	$(H,\ \Delta,\ XS,\ RS)$	
Heap	H	::=	$(TH,\ VH)$	
Pointer	p			
Type Heap	TH	::=	$p \overset{map}{\mapsto} ty$	
Heap Types	ty	::=	tn	(type name)
		|	$tn(p)$	(constructed type)
		|	$\overline{p}^{k}$	(tuple type)
		|	$p_1 \rightarrow p_2$	(function type)
		|	α	(type variable)
		|	$\langle\!\langle TE,\ p\rangle\!\rangle$	(polymorphic type)
Location	l			
Value Heap	VH	::=	$l \overset{map}{\mapsto} val$	
Heap Values	val	::=	$scon$	(special constant)
		|	con	(nullary constructor)
		|	$con(l)$	(unary constructor)
		|	$\overline{l}^{k}$	(tuple)
		|	$\langle\!\langle \Delta,\ \overline{x}^{k},\ E\rangle\!\rangle$	(closure)
		|	Ω	(dummy closure)
Environment	Δ	::=	$(TE,\ VE,\ DE,\ CE)$	
Type Variables	TE	::=	$\alpha \overset{map}{\mapsto} p$	
Lambda Variables	VE	::=	$x \overset{map}{\mapsto} (l,\ p)$	
Datatype Names	DE	::=	$tn \overset{map}{\mapsto} p$	
Constructors	CE	::=	$con \overset{map}{\mapsto} p$	
Exception Stack	XS	::=	$() \mid (l,\ p) \cdot XS$	
Result Stack	RS	::=	$() \mid p \cdot RS \mid (l,\ p) \cdot RS \mid \Delta \cdot RS$	

Figure 5.1: *Abstract Machine Syntax.*

Runtime Heap

The runtime heap is an abstraction for the *memory* of the abstract machine. The environment and stacks simply contain references into the heap and are therefore not considered part of the memory itself in our model. For convenience, we make a distinction in the heap between runtime types and runtime values. This division is not strictly necessary as we could store both values and types in the same heap. Indeed, in an actual implementation this would be advisable to avoid memory wastage. However, the separation of the values and types in the heap will make it easier to ensure that our definitions are correct, e.g. by ensuring that we do not accidentally treat values as types.

The runtime values, e.g. special constants and closures, are stored in the value heap VH, and the runtime types, corresponding to types and type schemes in the $\mathcal{M}\Lambda$ language, are stored in the type heap TH. We distinguish between pointers to runtime types p, and pointers to runtime values which we call locations l. The type heap and value heap are represented by finite-maps, as locations and pointers may be bound only once, i.e. there cannot be more than one value at a particular heap location. The separation of types and values in the heap is illustrated in Figure 5.2.

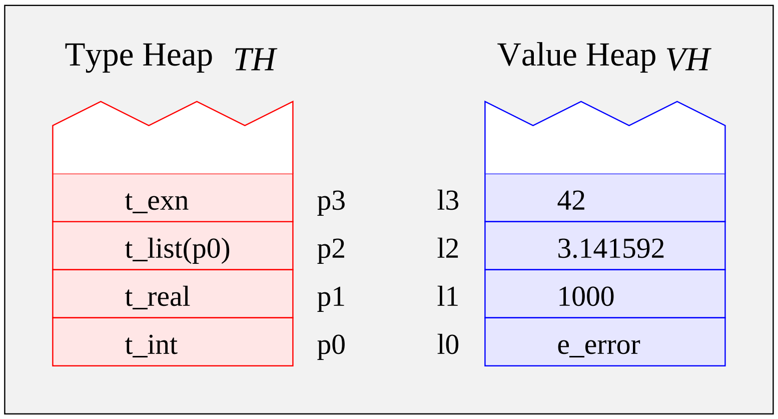


Figure 5.2: *An Example Runtime Heap H.*

A value *val* in the value heap will always be paired with a type *ty* in the type heap. We will use pairs of the form $(l,\ p)$ throughout this chapter to reference a value and its corresponding type. The reason for this pairing is that the type is required to determine the shape of the value. This will be explained in detail in Chapter 6 when we define garbage collection. Despite the pairing of values and types, it will generally be the case that the type heap will contain considerably less information than the value heap. This is due to the sharing of type information, e.g. only one integer type in the type heap is required for pairing every integer value in the value heap. It is worth noting that type information can be manipulated independently of the values, i.e. we can have types that do not correspond to values, but not vice-versa. This is because we require more flexibility when operating on types than we do with values, e.g. the instantiation of a polytype may require complex type manipulations.

It is important to note that there is no explicit notion of a memory *address* in our model of the heap. This is a deliberate abstraction which removes the need to consider explicit sizes of data items in the heap. As a result, it is not possible to perform such operations as pointer arithmetic, e.g. $l_1 + l_2$. The only operations permitted on pointers and locations are the comparison for equality, e.g. $l_1 = l_2$, and a dereferencing operation to perform the retrieval of the corresponding value and types from the heap : $VH(l) = val$ and $TH(p) = ty$. For notational convenience, we will refer only to the heap H, rather than the sub-heaps VH and TH, since the relevant sub-heap can always be determined from the context.

We will assume that values and types are allocated in a linear fashion through the heap. The management of the heap is discussed in more detail in Chapter 6, where a definition of garbage collection is presented. The following syntactic conventions are used for performing heap allocations: $H \uparrow l$ reserves and returns the next available location l in H, and $H[l \ \mapsto \ val]$ allocates value val at reserved location l in the value heap of H. Where multiple allocations are required we use the shorthand $H \uparrow (l^1, \ \ldots, \ l^k)$ which reserves and returns a sequence of locations $(l^1, \ \ldots, \ l^k)$, and $H[l^1 \ \mapsto \ val^1, \ \ldots, \ l^k \ \mapsto \ val^k]$ which allocates values $val^1, \ \ldots, \ val^k$ on the value heap, binding them to reserved locations. The notation for types is identical: $H \uparrow p$ reserves and returns the next available pointer p in H, and $H[p \ \mapsto \ ty]$ allocates type ty at p in the type heap of H.

There are no corresponding operations for removing values or types from the heap as this is achieved through garbage collection. However, the assignment of references, and the implementation of the fixed-point operator, necessitate a heap-update operation. Assignment uses the update operation $H[l_1 \overset{upd}{\mapsto} \text{c\_ref}(l_2)]$ to update the reference at l_1 to $\text{c\_ref}(l_2)$. This is clearly a trivial operation as it only requires the update of a single location. The fixed-point case is slightly more complex: $H[l \ \mapsto \ \Omega]$ allocates a dummy closure on the value heap bound to a fresh location l. This location can subsequently be updated with a mapping to a closure $H[l \overset{upd}{\mapsto} \langle\!\langle \Delta, \ \overline{x}^k, \ E \rangle\!\rangle]$. In an implementation of this operation, a suitably-sized area of the heap must be reserved to hold the closure.

Runtime Environment

The runtime environment Δ is essentially the dynamic counterpart of the static environment Γ defined in Chapter 3. The main purpose of the runtime environment is to record the allocation of $\mathcal{M}\Lambda$ variables and their corresponding values and types on the heap. We may consider the runtime environment as providing a view of the heap relevant to the program fragment being evaluated, i.e. the heap values and types corresponding to the bound variables which are currently in scope. The runtime environment also tracks the allocation of type variables, type names, and constructor types on the heap. This is an optimisation which we will use later in this chapter to prevent multiple allocations of the same types on the heap.

The runtime environment is composed of a four-tuple of sub-environments exactly as the static environment, i.e. a type variable environment TE, a lambda-variable environment VE, a datatype environment DE, and a constructor environment CE. However, we note that the definition of these sub-environments is different as we now map the variables, types, and constructors to explicit values and types on the heap. The notation for extending the runtime environment is exactly the same as for static environments: $\Delta[\theta]$ adds a binding θ to the environment, and $\Delta(\phi)$ retrieves the binding associated with ϕ from the environment. As before, the sub-environment in Δ is uniquely determined by the naming of θ and ϕ. There are no operations for removing bindings from the environment. However, unlike the heap, a copy of the current environment may be made at any time, e.g. by creating a closure. Thus, bindings can effectively be removed from the environment by reverting to an old copy of the environment.

Runtime Stacks

The runtime stacks store references into the heap. We do not store values or types directly on the stack. Scientific opinion is divided on whether this yields any performance advantage. In particular, memory management of the stack is considerably easier than the heap [Wil92], but stack allocation may itself be more costly [App87]. However, the avoidance of stack-allocated values and types will simplify the definition of garbage collection and code replacement since we can assume that all data is contained in the heap. The exception stack XS references exception handling functions (i.e. closures) and the result stack RS references temporary results during evaluation. The result stack is also used to hold temporary environments generated during let/letrec bindings.

5.2 Abstract Machine Semantics

The semantics of our abstract machine are defined by transitions between successive machine states. Evaluation of a program corresponds to a sequence of these transitions, beginning with an initial machine state, and terminating in a machine state representing the result of the computation. There is also the possibility of non-termination, corresponding to an infinite sequence of transitions, and early termination as a result of an uncaught exception, where the final machine state contains the exception packet.

As stated in the introduction, type-preserving evaluation becomes significantly more complex in the presence of polymorphism. The abstract machine presented in this chapter has undergone several revisions in order to provide efficient support for polymorphism. The first of these, presented in [WKG98], contained only monomorphic types. This definition is relatively straightforward as types are simply treated as static tags on the values, albeit allocated in a separate heap. With only monomorphic types, the abstract machine is entirely described by 21 rules. We originally intended this machine to be combined with a pre-processing stage, such as the one defined in [Els98], for converting the polymorphic types into monomorphic types. However, for reasons explained in Chapter 1, this technique was found to be incompatible with code-replacement.

Support for explicit polymorphism was introduced in a later revision of the abstract machine, presented in [Wal99]. The number of rules in the definition increased to 34. The definition presented in this paper is very clean, but not entirely satisfactory for an implementation of the language. In particular, each instantiation of a polymorphic type involves creating a completely new type in the type heap. This operation would be very costly to implement for all but the simplest type schemes, and would consume a large amount of heap space.

The latest revision of the abstract machine, presented in this chapter, contains an efficient scheme for manipulating polymorphic types, at the expense of increasing the complexity of the machine definition. The number of rules in this latest revision has increased to 47. However, an implementation of the machine should be significantly less costly, and considerably more space-efficient. Our scheme for manipulating polymorphic types is similar to the lazy scheme described in [Tol94], though the language on which we define our operation is somewhat different. We describe our scheme in detail in the following section, before presenting the abstract machine definition.

5.2.1 Runtime Types

It is clear that the syntax of the runtime types, defined in Figure 5.1, differs from the syntax of the static types defined previously in Chapter 2. Although the same classes of types are present in the runtime definition, we use pointers in place of syntactic categories to define the types. We therefore require a mechanism for converting static type definitions into heap-allocated runtime types. For monomorphic types, the $\tau\_alloc(\tau)$ function defined in Figure 5.3 performs this task. There is a separate rule for each of the types τ in $\mathcal{M}\Lambda$. Each rule is of the form: $(H_1,\ \Delta,\ XS,\ RS,\ \tau\_alloc(\tau)) \Rightarrow (H_2,\ \Delta,\ XS,\ p \cdot RS)$, where τ is the static type definition, H_1 is the heap before allocation, H_2 is the heap after allocation, and p is a pointer into H_2 to the newly allocated runtime type. The rules are defined recursively, with base cases for type names (Rule 5.1) and type variables (Rule 5.5). The example presented in Figure 5.4 illustrates the allocation of a type for a set-of-sets style representation of integer-labelled graphs.

$$\frac{}{(H,\ \Delta,\ XS,\ RS,\ \tau\_alloc(tn)) \Rightarrow (H,\ \Delta,\ XS,\ \Delta(tn) \cdot RS)} \tag{5.1}$$

$$\frac{(H_1,\ \Delta,\ XS,\ RS,\ \tau\_alloc(\overline{\tau}^k)) \Rightarrow (H_2,\ \Delta,\ XS,\ p_1 \cdot RS) \qquad H_2 \uparrow p_3}{(H_1,\ \Delta,\ XS,\ RS,\ \tau\_alloc(tn(\overline{\tau}^k))) \Rightarrow (H_2[p_3 \ \mapsto\ tn(p_1)],\ \Delta,\ XS,\ p_3 \cdot RS)} \tag{5.2}$$

$$\frac{\begin{array}{l}(H_1,\ \Delta,\ XS,\ RS_1,\ \tau\_alloc(\tau^1)) \Rightarrow (H_2,\ \Delta,\ XS,\ RS_2) \cdots \\ \quad (H_k,\ \Delta,\ XS,\ RS_k,\ \tau\_alloc(\tau^k)) \Rightarrow (H_{k+1},\ \Delta,\ XS,\ p_k \cdots p_1 \cdot RS_1) \\ H_{k+1} \uparrow p_{k+1}\end{array}}{\begin{array}{l}(H_1,\ \Delta,\ XS,\ RS_1,\ \tau\_alloc(\overline{\tau}^k)) \Rightarrow \\ \quad (H_{k+1}[p_{k+1} \ \mapsto\ (p_1,\ \ldots,\ p_k)],\ \Delta,\ XS,\ p_{k+1} \cdot RS_1)\end{array}} \tag{5.3}$$

$$\frac{\begin{array}{l}(H_1,\ \Delta,\ XS,\ RS,\ \tau\_alloc(\tau_1)) \Rightarrow (H_2,\ \Delta,\ XS,\ p_1 \cdot RS) \\ (H_2,\ \Delta,\ XS,\ p_1 \cdot RS,\ \tau\_alloc(\tau_2)) \Rightarrow (H_3,\ \Delta,\ XS,\ p_2 \cdot p_1 \cdot RS) \qquad H_3 \uparrow p_3\end{array}}{\begin{array}{l}(H_1,\ \Delta,\ XS,\ RS,\ \tau\_alloc(\tau_1 \rightarrow \tau_2)) \Rightarrow \\ \quad (H_3[p_3 \ \mapsto\ p_1 \rightarrow p_2],\ \Delta,\ XS,\ p_3 \cdot RS)\end{array}} \tag{5.4}$$

$$\frac{}{(H,\ \Delta,\ XS,\ RS,\ \tau\_alloc(\alpha)) \Rightarrow (H,\ \Delta,\ XS,\ \Delta(\alpha) \cdot RS)} \tag{5.5}$$

Figure 5.3: *Monotype Allocation.*

Standard ML Type: `type intgraph = ((string * int) list) * (string -> (string * int) list)`

$\mathcal{M}\Lambda$ **Static Type:** $(\text{t\_list}(\text{t\_string},\ \text{t\_int}),\ \text{t\_list}((\text{t\_string} \rightarrow (\text{t\_string},\ \text{t\_int}))))$

$\mathcal{M}\Lambda$ **Dynamic Type Allocation:**

$$\underbrace{\tau\_\mathit{alloc}(\text{t\_string}) \Rightarrow p_2 \qquad \tau\_\mathit{alloc}(\text{t\_int}) \Rightarrow p_1 \qquad p_4 \mapsto (p_2,\ p_1)}_{(5.3)}$$

$$\underbrace{\tau\_\mathit{alloc}(\text{t\_string}) \Rightarrow p_2 \qquad \tau\_\mathit{alloc}(\text{t\_int}) \Rightarrow p_1 \qquad p_5 \mapsto (p_2,\ p_1)}_{(5.3)} \qquad \underbrace{\tau\_\mathit{alloc}(\text{t\_string}) \Rightarrow p_2 \qquad \tau\_\mathit{alloc}((\text{t\_string},\ \text{t\_int})) \Rightarrow p_4 \qquad p_6 \mapsto p_2 \rightarrow p_4}_{(5.4)}$$

$$\underbrace{\tau\_\mathit{alloc}((\text{t\_string},\ \text{t\_int})) \Rightarrow p_5 \qquad p_7 \mapsto \text{t\_list}(p_5)}_{(5.2)} \qquad \underbrace{\tau\_\mathit{alloc}((\text{t\_string} \rightarrow (\text{t\_string},\ \text{t\_int}))) \Rightarrow p_6 \qquad p_8 \mapsto \text{t\_list}(p_6)}_{(5.2)}$$

$$\underbrace{\tau\_\mathit{alloc}(\text{t\_list}(\text{t\_string},\ \text{t\_int})) \Rightarrow p_7 \qquad \tau\_\mathit{alloc}(\text{t\_list}((\text{t\_string} \rightarrow (\text{t\_string},\ \text{t\_int})))) \Rightarrow p_8 \qquad p_9 \mapsto (p_7,\ p_8)}_{(5.3)}$$

$$\tau\_\mathit{alloc}((\text{t\_list}(\text{t\_string},\ \text{t\_int}),\ \text{t\_list}((\text{t\_string} \rightarrow (\text{t\_string},\ \text{t\_int}))))) \Rightarrow p_9$$

Figure 5.4: *Runtime Type Allocation Example.*

A number of simplifications have been made in the presentation of the example: the rules are given in the form $\tau\_alloc(\tau) \Rightarrow p$; the heap, environment and stacks are not shown, and the notation $p \mapsto ty$ is used to indicate the binding of a new type ty to a new pointer p in the heap. We assume that the heap initially contains the bindings $\{p_1 \mapsto \text{t\_int},\ p_2 \mapsto \text{t\_string}\}$ and the environment contains the bindings $\{\text{t\_int} \mapsto p_1,\ \text{t\_string} \mapsto p_2\}$. The braces are numbered according to the rule used at each stage. The rules should be read in bottom-up left-right order. The result of the allocation is the pointer p_9 to the heap-allocated type.

We now turn our attention to the allocation of type-schemes. As stated earlier in this chapter, we require a space-efficient representation, and a cheap implementation of instantiation. If we consider the type scheme for the polymorphic identity function $\forall\alpha.\ \alpha \rightarrow \alpha$, a straightforward representation is a runtime type of the form $\forall p.\ p \rightarrow p$, where p is a pointer to a heap allocated type variable α. It appears that we may perform instantiation cheaply by simply replacing the type variable with a type, e.g. to generate the integer identity function we perform a heap update $H[p \overset{upd}{\mapsto} \text{t\_int}]$. This technique appears to be space efficient and cheap. However, with a little thought it is clear that is it flawed. By updating the type variable with a type, we permit only one instance of a type scheme. If we use the update operation to perform multiple instantiations, we will overwrite any previous instances with our new type.

We may avoid the update problem by considering the runtime type scheme as a template. A new instance may therefore be generated by recursively traversing the type scheme and filling in the actual types in place of the type variables. This scheme was used successfully in an earlier definition [Wal99]. However, we now have neither a cheap implementation or a space-efficient representation; the recursive traversal operation will be costly to implement, particularly for complex types, and each instance will be separately heap-allocated. When we define garbage collection and replacement, we will need to examine every instance of a type scheme on the heap. Yet, we are only concerned with the types of the variables which are live at the point of collection and replacement. Thus, a great deal of the instantiation work is completely unnecessary.

We have therefore developed an alternative scheme which performs a deferred style of instantiation. We represent a type scheme as a construct which we call a type closure containing a type-variable environment, and a type pointer: $\langle\!\langle TE,\ p\rangle\!\rangle$. The type variable environment maps type variables to types, and the type pointer points to the type-scheme template as before. For example, the identity function type is represented as $\langle\!\langle \{\alpha \mapsto \alpha\},\ p \mapsto \alpha \rightarrow \alpha \rangle\!\rangle$. Instantiation is now achieved by creating a new closure containing an updated type environment, e.g. $\langle\!\langle \{\alpha \mapsto \text{t\_int}\},\ p \mapsto \alpha \rightarrow \alpha \rangle\!\rangle$. Note that the pointer to the template does not require updating. Thus, this operation can be implemented very cheaply. However, when one operates on the type scheme, it is necessary to consult the associated type environment to interpret the type variables. Thus, the work performed by instantiation is deferred until it is actually required. We define deferred instantiation in Figure 5.5. The *instance* function is invoked with a pointer p_σ to a type closure, and a pointer p_τ to a tuple of type pointers which are bound to type variables in the type environment. The result is a new pointer p_4 to the new type closure on the heap. This technique is clearly far more space-efficient than allocating separate types for each instance. However, the abstract machine definition becomes necessarily more complex when referring to polymorphic types.

$$\frac{H(p_\sigma) = \langle\!\langle \{\alpha^1 \mapsto p_1^1,\ \ldots,\ \alpha^k \mapsto p_1^k\},\ p_2 \rangle\!\rangle \qquad H(p_\tau) = \overline{p_3}^{\,k} \qquad H \uparrow p_4}{\begin{array}{l}(H,\ \Delta,\ \mathit{XS},\ \mathit{RS},\ \mathit{instance}(p_\sigma,\ p_\tau)) \Rightarrow \\ \quad (H[p_4 \mapsto \langle\!\langle \{\alpha^1 \mapsto p_3^1,\ \ldots,\ \alpha^k \mapsto p_3^k\},\ p_2 \rangle\!\rangle],\ \Delta,\ \mathit{XS},\ p_4 \cdot \mathit{RS})\end{array}} \tag{5.6}$$

Figure 5.5: *Deferred Type Instantiation.*

We are now in a position to define the runtime allocation of type-schemes as type closures. The allocation is performed by two functions: $\alpha\_alloc(\sigma)$ which allocates the quantified type variables, and $\sigma\_alloc(\sigma)$ which generates the type closures. Note that the monomorphic rule for type variables 5.5 returns a pointer obtained from the environment Δ. Thus, in order to create a polymorphic type, it is first necessary to extend the environment with the type variables, which in turn requires the type variables to be allocated on the heap. These steps are performed by the function defined in Figure 5.6 which has the form: $(H_1,\ \Delta_1,\ \mathit{XS},\ \mathit{RS},\ \alpha\_alloc(\sigma)) \Rightarrow (H_2,\ \Delta_2,\ \mathit{XS},\ \mathit{RS})$.

$$\frac{}{(H,\ \Delta,\ \mathit{XS},\ \mathit{RS},\ \alpha\_alloc(\tau)) \Rightarrow (H,\ \Delta,\ \mathit{XS},\ \mathit{RS})} \tag{5.7}$$

$$\frac{H \uparrow (p^1,\ \ldots,\ p^k)}{\begin{array}{l}(H,\ \Delta,\ \mathit{XS},\ \mathit{RS},\ \alpha\_alloc(\forall\, \overline{\alpha}^k.\ \tau)) \Rightarrow \\ \quad (H[p^1 \mapsto \alpha^1,\ \ldots,\ p^k \mapsto \alpha^k],\ \Delta[\alpha^1 \mapsto p^1,\ \ldots,\ \alpha^k \mapsto p^k],\ \mathit{XS},\ \mathit{RS})\end{array}} \tag{5.8}$$

Figure 5.6: *Type-Variable Allocation.*

There are two definitions of the σ_alloc function. The first of these (Rule 5.10) has the form $(H_1,\ \Delta,\ XS,\ RS,\ \sigma\_alloc(\sigma)) \Rightarrow (H_2,\ \Delta,\ XS,\ p \cdot RS)$, where p is a pointer to the resulting type closure. This rule takes a type scheme as an argument and allocates the type variables using α_alloc before allocating the quantified type using τ_alloc, and constructing the type closure. Note that the environment Δ_1 is stored on the result stack at the beginning of the rule and restored at the end. This removes the type variables from the main environment once they are present in the environment of the type closure. In the monomorphic case (Rule 5.9) the type is simply passed to τ_alloc and no type closure is constructed.

The second definition of σ_alloc (Rule 5.12) takes a type scheme and a pointer as arguments: $(H_1,\ \Delta,\ XS,\ RS,\ \sigma\_alloc(\sigma,\ p_1)) \Rightarrow (H_2,\ \Delta,\ XS,\ p_2 \cdot RS)$. This definition assumes that the type variables have already been allocated and are present in the environment, and the quantified type τ has been allocated and is referenced by the pointer p_1. This form is used for binding values to polymorphic types inside the let expressions (recall from Chapter 3 that expressions always have monomorphic types). In this case, a type closure is simply created from the type variables in the environment, and the supplied pointer p_1. The monomorphic case (Rule 5.11) simply returns the pointer as it is not necessary to create a type closure.

$$\frac{(H_1,\ \Delta,\ XS,\ RS_1,\ \tau\_alloc(\tau)) \Rightarrow (H_2,\ \Delta,\ XS,\ RS_2)}{(H_1,\ \Delta,\ XS,\ RS_1,\ \sigma\_alloc(\tau)) \Rightarrow (H_2,\ \Delta,\ XS,\ RS_2)} \tag{5.9}$$

$$\frac{\begin{array}{l}(H_1,\ \Delta_1,\ XS,\ \Delta_1 \cdot RS_1,\ \alpha\_alloc(\forall\, \overline{\alpha}^k.\ \tau)) \Rightarrow (H_2,\ \Delta_2,\ XS,\ RS_2)\\ (H_2,\ \Delta_2,\ XS,\ RS_2,\ \tau\_alloc(\tau)) \Rightarrow (H_3,\ \Delta_2,\ XS,\ p_2 \cdot \Delta_1 \cdot RS_1)\\ H_3 \uparrow p_3\end{array}}{\begin{array}{l}(H_1,\ \Delta_1,\ XS,\ RS_1,\ \sigma\_alloc(\forall\, \overline{\alpha}^k.\ \tau)) \Rightarrow\\ \quad (H_3[p_3 \ \mapsto\ \langle\!\langle \{\alpha^1 \ \mapsto\ \Delta_2(\alpha^1),\ \ldots,\ \alpha^k \ \mapsto\ \Delta_2(\alpha^k)\},\ p_2 \rangle\!\rangle],\ \Delta_1,\ XS,\ p_3 \cdot RS_1)\end{array}} \tag{5.10}$$

$$\frac{}{(H,\ \Delta,\ XS,\ RS,\ \sigma\_alloc(\tau,\ p)) \Rightarrow (H,\ \Delta,\ XS,\ p \cdot RS)} \tag{5.11}$$

$$\frac{H \uparrow p_2}{\begin{array}{l}(H,\ \Delta,\ XS,\ RS,\ \sigma\_alloc(\forall\, \overline{\alpha}^k.\ \tau,\ p_1)) \Rightarrow\\ \quad (H[p_2 \ \mapsto\ \langle\!\langle \{\alpha_1 \ \mapsto\ \Delta(\alpha^1),\ \ldots,\ \alpha^k \ \mapsto\ \Delta(\alpha^k)\},\ p_1 \rangle\!\rangle],\ \Delta,\ XS,\ p_2 \cdot RS)\end{array}} \tag{5.12}$$

Figure 5.7: *Type-Scheme Allocation.*

5.2.2 Machine Transitions

The initial machine state, corresponding to the initial dynamic basis of Standard ML, is given in Figure 5.8. The initial type heap contains the types of the special constants, built-in functions, datatypes, constructors, and exceptions. The value heap contains closures which represent the built-in functions on the special constants. For brevity we have again only presented the functions on the integers. The actual code for these functions is denoted by the lower-case name e.g. abs_i, as we cannot directly express these operations as $\mathcal{M}\Lambda$ expressions. We assume that an actual compiler would substitute these with the necessary machine-level instructions. The variable environment maps the identifiers of these functions to their heap types and closures. For later convenience we provide an additional variable called UNIT. The datatype environment and constructor environment map the datatype and constructor identifiers to their types in the type heap. The datatype environment is primarily an optimisation to prevent the repeated allocation of type names. The exception stack and result stack are initially empty.

$$M = (H,\ \Delta,\ (),\ ()) \qquad H = (TH,\ VH) \qquad \Delta = (\emptyset,\ VE,\ DE,\ CE)$$

$$TH = \left\{ \begin{array}{l} p_1 \mapsto \text{t\_unit},\ p_2 \mapsto \text{t\_int},\ p_3 \mapsto \text{t\_word},\ p_4 \mapsto \text{t\_real}, \\ p_5 \mapsto \text{t\_char},\ p_6 \mapsto \text{t\_string},\ p_7 \mapsto \text{t\_bool},\ p_8 \mapsto \text{t\_exn}, \\ p_9 \mapsto (p_2,\ p_2),\ p_{10} \mapsto p_2 \to p_2,\ p_{11} \mapsto p_9 \to p_2,\ p_{12} \mapsto p_9 \to p_7, \\ p_{13} \mapsto \alpha,\ p_{14} \mapsto \text{t\_list}(p_{13}),\ p_{15} \mapsto \langle\!\langle \{\alpha \mapsto p_{13}\},\ p_{14} \rangle\!\rangle, \\ p_{16} \mapsto (p_{13},\ p_{14}),\ p_{17} \mapsto p_{16} \to p_{14},\ p_{18} \mapsto \langle\!\langle \{\alpha \mapsto p_{13}\},\ p_{17} \rangle\!\rangle, \\ p_{19} \mapsto \text{t\_ref}(p_{13}),\ p_{20} \mapsto p_{13} \to p_{19},\ p_{21} \mapsto \langle\!\langle \{\alpha \mapsto p_{13}\},\ p_{20} \rangle\!\rangle \end{array} \right\}$$

$$VH = \left\{ \begin{array}{l} l_1 \mapsto \text{scon}_{unit},\ l_2 \mapsto \langle\!\langle \emptyset,\ (x_1),\ abs_i \rangle\!\rangle,\ l_3 \mapsto \langle\!\langle \emptyset,\ (x_1),\ neg_i \rangle\!\rangle, \\ l_4 \mapsto \langle\!\langle \emptyset,\ (x_1,\ x_2),\ div_i \rangle\!\rangle,\ l_5 \mapsto \langle\!\langle \emptyset,\ (x_1,\ x_2),\ mod_i \rangle\!\rangle, \\ l_6 \mapsto \langle\!\langle \emptyset,\ (x_1,\ x_2),\ mul_i \rangle\!\rangle,\ l_7 \mapsto \langle\!\langle \emptyset,\ (x_1,\ x_2),\ add_i \rangle\!\rangle, \\ l_8 \mapsto \langle\!\langle \emptyset,\ (x_1,\ x_2),\ sub_i \rangle\!\rangle,\ l_9 \mapsto \langle\!\langle \emptyset,\ (x_1,\ x_2),\ lt_i \rangle\!\rangle, \\ l_{10} \mapsto \langle\!\langle \emptyset,\ (x_1,\ x_2),\ gt_i \rangle\!\rangle,\ l_{11} \mapsto \langle\!\langle \emptyset,\ (x_1,\ x_2),\ leq_i \rangle\!\rangle, \\ l_{12} \mapsto \langle\!\langle \emptyset,\ (x_1,\ x_2),\ geq_i \rangle\!\rangle,\ l_{13} \mapsto \langle\!\langle \emptyset,\ (x_1,\ x_2),\ eq_i \rangle\!\rangle \end{array} \right\}$$

$$VE = \left\{ \begin{array}{l} \text{UNIT} \mapsto (l_1,\ p_1),\ \text{ABS}_i \mapsto (l_2,\ p_{10}),\ \text{NEG}_i \mapsto (l_3,\ p_{10}), \\ \text{DIV}_i \mapsto (l_4,\ p_{11}),\ \text{MOD}_i \mapsto (l_5,\ p_{11}),\ \text{MUL}_i \mapsto (l_6,\ p_{11}), \\ \text{ADD}_i \mapsto (l_7,\ p_{11}),\ \text{SUB}_i \mapsto (l_8,\ p_{11}),\ \text{LT}_i \mapsto (l_9,\ p_{12}), \\ \text{GT}_i \mapsto (l_{10},\ p_{12}),\ \text{LEQ}_i \mapsto (l_{11},\ p_{12}),\ \text{GEQ}_i \mapsto (l_{12},\ p_{12}), \\ \text{EQ}_i \mapsto (l_{13},\ p_{12}) \end{array} \right\}$$

$$DE = \left\{ \begin{array}{l} \text{t\_unit} \mapsto p_1,\ \text{t\_int} \mapsto p_2,\ \text{t\_word} \mapsto p_3,\ \text{t\_real} \mapsto p_4, \\ \text{t\_char} \mapsto p_5,\ \text{t\_string} \mapsto p_6,\ \text{t\_bool} \mapsto p_7,\ \text{t\_exn} \mapsto p_8 \end{array} \right\}$$

$$CE = \left\{ \begin{array}{l} \text{c\_true} \mapsto p_7,\ \text{c\_false} \mapsto p_7,\ \text{c\_nil} \mapsto p_{15},\ \text{c\_cons} \mapsto p_{18}, \\ \text{c\_ref} \mapsto p_{21},\ \text{e\_match} \mapsto p_8,\ \text{e\_bind} \mapsto p_8,\ \text{e\_overflow} \mapsto p_8 \end{array} \right\}$$

Figure 5.8: *Initial Machine State.*

We may now define the dynamic semantics of the sequential fragment of $\mathcal{M}\Lambda$. As with the functions for type allocation and instantiation, we will enumerate the entire machine state in each rule. Evaluation of an $\mathcal{M}\Lambda$ program begins with Rule 5.13. The datatypes and exceptions are evaluated in turn resulting in an extended heap and environment. The main body of the program, given by the expression E, is evaluated in this extended environment to yield a result on the result stack. As with the typing judgements, there are many similarities between the rules. Therefore, we will just highlight the main features as comments.

In order to support garbage collection and replacement, we impose a number of restrictions on our rules. As the garbage collector is a copying collector, we must assume that the entire machine state may change during collection. Thus, each time we evaluate an expression, we assume that a new machine state is generated: $(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ E) \Rightarrow (H_2,\ \Delta_2,\ XS_2,\ (l,\ p) \cdot RS_2)$, i.e. even though RS_2 will contain the same items as RS_1. In Chapter 6 (Rule 6.1) we define precisely when a collection may occur. Informally, a collection can only happen at the beginning of the evaluation of an expression (or sub-expression), i.e. immediately after we start the evaluation but before any work is done. Consequently, we must take care to store any live pointers on the result stack before evaluating any sub-expressions. This is neatly illustrated in Rule 5.38. Since we have adopted a left-to-right evaluation strategy, we must evaluate the function expression E_1, before evaluating the argument expression E_2. Since garbage collection may occur between the evaluation of E_1 and E_2, we must store the pair $(l_1,\ p_1)$, which references the function closure, on the result stack before evaluating E_2 or it may be discarded. After the evaluation of E_2, we write this pair as $(l_1',\ p_1')$ to indicate that the pair may have been updated by garbage collection. We will also use this notation for the result stack, e.g. the result stack RS_1'' has the same contents as the result stack RS_1 but may have been garbage collected twice.

A number of abstract machines for ML-like languages, e.g. the $\lambda_{\mathrm{gc}}^{\rightarrow\forall}$ abstract machine [MH96], adopt a restricted *named-form* representation of types and values. This restriction amounts to the requirement that every step in the evaluation be bound to a variable, i.e. every expression is a **let**. This is largely a technical convenience as it allows a number of simplifications to the abstract machine definition, e.g. the type of an expression can easily be obtained from the types of it's sub-expressions. In the $\mathcal{M}\Lambda$ abstract machine, we always return a pair $(l,\ p)$ on the result stack as the result of every evaluation. We call this *implicit* named-form, as it gives all the benefits of the named form representation, without cluttering the environment with superfluous variable declarations.

Programs

$$\frac{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ \overline{\overline{D}}) \Rightarrow (H_2,\ \Delta_2,\ XS_1,\ RS_1)\\ (H_2,\ \Delta_2,\ XS_1,\ RS_1,\ \overline{\overline{X}}) \Rightarrow (H_3,\ \Delta_3,\ XS_1,\ RS_1)\\ (H_3,\ \Delta_3,\ XS_1,\ RS_1,\ E) \Rightarrow (H_4,\ \Delta_4,\ XS_2,\ RS_2)\end{array}}{(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ (\overline{\overline{D}},\ \overline{\overline{X}},\ E)) \Rightarrow (H_4,\ \Delta_4,\ XS_2,\ RS_2)} \tag{5.13}$$

Datatype Declarations

$$\frac{}{(H,\ \Delta,\ XS,\ RS,\ \emptyset) \Rightarrow (H,\ \Delta,\ XS,\ RS)} \tag{5.14}$$

$$\frac{\begin{array}{l}(H_1,\ \Delta_1,\ XS,\ RS,\ D^1) \Rightarrow (H_2,\ \Delta_2,\ XS,\ RS)\ \cdots\\ \quad (H_k,\ \Delta_k,\ XS,\ RS,\ D^k) \Rightarrow (H_{k+1},\ \Delta_{k+1},\ XS,\ RS)\end{array}}{(H_1,\ \Delta_1,\ XS,\ RS,\ \{D^1,\ \ldots,\ D^k\}) \Rightarrow (H_{k+1},\ \Delta_{k+1},\ XS,\ RS)} \tag{5.15}$$

$$\frac{\begin{array}{l}H_1 \uparrow p_1\\ (H_1[p_1 \mapsto tn],\ \Delta_1[tn \mapsto p_1],\ XS,\ RS_1,\ \tau\_alloc(ctype(C^1,\ tn))) \Rightarrow\\ \quad (H_2,\ \Delta_2,\ XS,\ RS_2)\ \cdots\\ \qquad (H_k,\ \Delta_k,\ XS,\ RS_k,\ \tau\_alloc(ctype(C^k,\ tn))) \Rightarrow\\ \qquad\quad (H_{k+1},\ \Delta_{k+1},\ XS,\ p_2^k \cdots p_2^1 \cdot RS_1)\end{array}}{\begin{array}{l}(H_1,\ \Delta_1,\ XS,\ RS_1,\ \textbf{datatype}\ tn\ \textbf{of}\ \{C^1,\ \ldots,\ C^k\}) \Rightarrow\\ \quad (H_{k+1},\ \Delta_{k+1}[cname(C^1) \mapsto p_2^1,\ \ldots,\ cname(C^k) \mapsto p_2^k],\ XS,\ RS_1)\end{array}} \tag{5.16}$$

Comment: (Rule 5.16) Evaluation of a monomorphic datatype declaration begins with the allocation of the datatype name *tn* on the heap, and a corresponding mapping $tn \mapsto p_1$ in the environment. We reuse this mapping in the allocation of monomorphic constructor types (Rule 5.1). The constructor types are then allocated on the heap using the τ_alloc function, and the environment is extended with a set of mappings from each of the constructors *con* to their heap allocated types. The functions *ctype* (5.17) and *cname* (5.18) return the static type and name for a constructor respectively

$$\begin{array}{lcl} ctype(con,\ tn) & = & tn\\ ctype((con,\ \tau),\ tn) & = & \tau \rightarrow tn\\ ctype(con,\ (\overline{\alpha}^k,\ tn)) & = & \forall\, \overline{\alpha}^k.\ tn(\overline{\alpha}^k)\\ ctype((con,\ \tau),\ (\overline{\alpha}^k,\ tn)) & = & \forall\, \overline{\alpha}^k.\ \tau \rightarrow tn(\overline{\alpha}^k) \end{array} \tag{5.17}$$

$$\begin{array}{lcl} cname(con) & = & con\\ cname((con,\ \tau)) & = & con \end{array} \tag{5.18}$$

$$\frac{\begin{array}{l}(H_1,\ \Delta_1,\ XS,\ RS_1,\ \sigma\_alloc(ctype(C^1,\ (\overline{\alpha}^k,\ tn)))) \Rightarrow \\ \quad (H_2,\ \Delta_2,\ XS,\ RS_2)\ \cdots \\ \qquad (H_k,\ \Delta_k,\ XS,\ RS_k,\ \sigma\_alloc(ctype(C^k,\ (\overline{\alpha}^k,\ tn)))) \Rightarrow \\ \qquad\quad (H_{k+1},\ \Delta_{k+1},\ XS,\ p_1^k \cdots p_1^1 \cdot RS_1)\end{array}}{\begin{array}{l}(H_1,\ \Delta_1,\ XS,\ RS_1,\ \textbf{datatype}\ (\overline{\alpha}^k, tn)\ \textbf{of}\ \{C^1,\ \ldots,\ C^k\}) \Rightarrow \\ \quad (H_{k+1},\ \Delta_{k+1}[cname(C^1)\ \mapsto\ p_1^1,\ \ldots,\ cname(C^k)\ \mapsto\ p_1^k],\ XS,\ RS_1)\end{array}} \tag{5.19}$$

Comment: (Rule 5.19) The evaluation of a polymorphic datatype is very similar to the monomorphic case. However, the datatype name *tn* is not allocated on the heap as it is not used in the allocation of polymorphic constructor types (Rule 5.2). Also, note that it is necessary to use the *σ_alloc* function in this rule for the allocation of the constructor types.

Exception Declarations

$$\frac{}{(H,\ \Delta,\ XS,\ RS,\ \emptyset) \Rightarrow (H,\ \Delta,\ XS,\ RS)} \tag{5.20}$$

$$\frac{\begin{array}{l}(H_1,\ \Delta_1,\ XS,\ RS_1,\ \tau\_alloc(ctype(C^1,\ \mathrm{t\_exn}))) \Rightarrow (H_2,\ \Delta_2,\ XS,\ RS_2)\ \cdots \\ \quad (H_k,\ \Delta_k,\ XS,\ RS_k,\ \tau\_alloc(ctype(C^k,\ \mathrm{t\_exn}))) \Rightarrow \\ \qquad (H_{k+1},\ \Delta_{k+1},\ XS,\ p^k \cdots p^1 \cdot RS_1)\end{array}}{\begin{array}{l}(H_1,\ \Delta_1,\ XS,\ RS_1,\ \{\textbf{exception}\ C^1,\ \ldots,\ \textbf{exception}\ C^k\}) \Rightarrow \\ \quad (H_{k+1},\ \Delta_{k+1}[cname(C^1)\ \mapsto\ p^1,\ \ldots,\ cname(C^k)\ \mapsto\ p^k],\ XS,\ RS_1)\end{array}} \tag{5.21}$$

Comment: (Rule 5.21) The evaluation of the exception declarations essentially corresponds to the evaluation of a single monomorphic datatype (Rule 5.16). However, the type name t_exn is not allocated on the heap as it is present in the initial machine state (Figure 5.8). Note that all of the exception declarations are treated as constructors of the t_exn datatype.

Values

$$\frac{H \uparrow l}{\begin{array}{l}(H,\ \Delta,\ XS,\ RS,\ \textbf{scon}\ scon) \Rightarrow \\ \quad (H[l\ \mapsto\ scon],\ \Delta,\ XS,\ (l,\ \Delta(tn_{scon})) \cdot RS)\end{array}} \tag{5.22}$$

Comment: (Rule 5.22) Special constants are allocated directly on the heap. The type pointer for the special constant is obtained from the environment. The type tn_{scon} is trivially determined from the lexical structure of *scon*.

$$\frac{}{(H,\ \Delta,\ XS,\ RS,\ \textbf{var}\ x) \Rightarrow (H,\ \Delta,\ XS,\ \Delta(x) \cdot RS)} \tag{5.23}$$

$$\frac{\begin{array}{l}\Delta(x) = (l_1,\ p_1)\\ (H_1,\ \Delta,\ XS,\ RS,\ \tau\_alloc(\overline{\tau}^k)) \Rightarrow (H_2,\ \Delta,\ XS,\ p_2 \cdot RS)\\ (H_2,\ \Delta,\ XS,\ RS,\ instance(p_1,\ p_2)) \Rightarrow (H_3,\ \Delta,\ XS,\ p_3 \cdot RS)\end{array}}{(H_1,\ \Delta,\ XS,\ RS,\ \textbf{var}\ (\overline{\tau}^k,\ x)) \Rightarrow (H_3,\ \Delta,\ XS,\ (l_1,\ p_3) \cdot RS)} \tag{5.24}$$

Comment: (Rule 5.24) The instantiation of a polymorphic variable is performed in three steps. A reference p_1 to the type scheme of the variable is obtained from the environment. The types $\overline{\tau}^k$ which will replace the type variables are allocated on the heap at p_2. Finally, an instance is created by invoking $instance(p_1,\ p_2)$. Recall that we use a deferred instantiation scheme. Thus, the result of the instantiation p_3 will be a reference to a type closure.

$$\frac{(H_1,\ \Delta,\ XS,\ RS,\ \tau\_alloc(\tau_1 \rightarrow \tau_2)) \Rightarrow (H_2,\ \Delta,\ XS,\ p \cdot RS) \qquad H_2 \uparrow l}{\begin{array}{l}(H_1,\ \Delta,\ XS,\ RS,\ \textbf{fn}\ (x,\ \tau_1 \rightarrow \tau_2)\ =\ E) \Rightarrow\\ \quad (H_2[l\ \mapsto\ \langle\!\langle \Delta,\ (x),\ E \rangle\!\rangle],\ \Delta,\ XS,\ (l,\ p) \cdot RS)\end{array}} \tag{5.25}$$

$$\frac{(H_1,\ \Delta,\ XS,\ RS,\ \tau\_alloc(\overline{\tau_1}^k \rightarrow \tau_2)) \Rightarrow (H_2,\ \Delta,\ XS,\ p \cdot RS) \qquad H_2 \uparrow l}{\begin{array}{l}(H_1,\ \Delta,\ XS,\ RS,\ \textbf{fn}\ (\overline{x}^k,\ \overline{\tau_1}^k \rightarrow \tau_2)\ =\ E) \Rightarrow\\ \quad (H_2[l\ \mapsto\ \langle\!\langle \Delta,\ \overline{x}^k,\ E \rangle\!\rangle],\ \Delta,\ XS,\ (l,\ p) \cdot RS)\end{array}} \tag{5.26}$$

Comment: (Rules 5.25 and 5.26) These rules allocate a new closure on the value heap. The second form (5.26) is for functions which take multiple arguments. The closure consists of a copy of the environment, a sequence of variables to be bound to the function parameters, and an expression for the body of the function.

Value Constructors

$$\frac{H \uparrow l}{(H,\ \Delta,\ XS,\ RS,\ \textbf{con}\ con) \Rightarrow (H[l_1\ \mapsto\ con],\ \Delta,\ XS,\ (l,\ \Delta(con)) \cdot RS)} \tag{5.27}$$

Comment: (Rule 5.27) A monomorphic nullary constructor (e.g. c_true) is allocated directly on the heap $l\ \mapsto\ con$. The type of the constructor is obtained from the environment $\Delta(con)$.

$$\frac{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ E) \Rightarrow (H_2,\ \Delta_2,\ XS_2,\ (l_1,\ p_1)\cdot RS'_1) \\ H_2(\Delta_2(con)) = p_3 \rightarrow p_4 \qquad H_2 \uparrow l_2\end{array}}{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ \mathbf{con}\ (con,\ E)) \Rightarrow \\ \quad (H_2[l_2\ \mapsto\ con(l_1)],\ \Delta_2,\ XS_2,\ (l_2,\ p_4)\cdot RS'_1)\end{array}} \tag{5.28}$$

Comment: (Rule 5.28) A unary constructor is built by evaluating an expression E, which forms the body of the constructed value. The type of the unary constructor $\Delta_2(con)$ is the function type $p_3 \rightarrow p_4$. Note that the function result type p_4 is returned as the type of the constructed value.

$$\frac{\begin{array}{l}(H_1,\ \Delta,\ XS,\ RS,\ \tau\_alloc(\overline{\tau}^k)) \Rightarrow (H_2,\ \Delta,\ XS,\ p_1\cdot RS) \\ H_2(\Delta(con)) = \langle\!\langle\{\alpha^1\ \mapsto\ p_2^1,\ \ldots,\ \alpha^k\ \mapsto\ p_2^k\},\ p_3\rangle\!\rangle \qquad H_2(p_1) = \overline{p_4}^k \\ H_2 \uparrow l_1 \qquad H_2 \uparrow p_5\end{array}}{\begin{array}{l}(H_1,\ \Delta,\ XS,\ RS,\ \mathbf{con}\ (con,\ \overline{\tau}^k)) \Rightarrow \\ \quad (H_2[l_1\ \mapsto\ con][p_5\ \mapsto\ \langle\!\langle\{\alpha^1\ \mapsto\ p_4^1,\ \ldots,\ \alpha^k\ \mapsto\ p_4^k\},\ p_3\rangle\!\rangle], \\ \qquad \Delta,\ XS,\ (l_1,\ p_5)\cdot RS)\end{array}} \tag{5.29}$$

Comment: (Rule 5.29) The construction of a nullary polymorphic constructor (e.g. c_nil) is essentially the same as the *instance* function defined in Figure 5.5, given $p_\sigma = \Delta(con)$ and $p_\tau = p_1$.

$$\frac{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ E) \Rightarrow (H_2,\ \Delta_2,\ XS_2,\ (l_1,\ p_1)\cdot RS'_1) \\ (H_2,\ \Delta_2,\ XS_2,\ (l_1,\ p_1)\cdot RS'_1,\ \tau\_alloc(\overline{\tau}^k)) \Rightarrow \\ \quad (H_3,\ \Delta_2,\ XS_2,\ p_2\cdot(l_1,\ p_1)\cdot RS'_1) \\ H_3(\Delta(con)) = \langle\!\langle\{\alpha^1\ \mapsto\ p_3^1,\ \ldots,\ \alpha^k\ \mapsto\ p_3^k\},\ p_4\rangle\!\rangle \qquad H_3(p_2) = \overline{p_5}^k \\ H_3(p_4) = p_6 \rightarrow p_7 \qquad H_3 \uparrow l_2 \qquad H_3 \uparrow p_8\end{array}}{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ \mathbf{con}\ (con,\ \overline{\tau}^k,\ E)) \Rightarrow \\ \quad (H_3[l_2\ \mapsto\ con(l_1)][p_8\ \mapsto\ \langle\!\langle\{\alpha^1\ \mapsto\ p_5^1,\ \ldots,\ \alpha^k\ \mapsto\ p_5^k\},\ p_7\rangle\!\rangle], \\ \qquad \Delta_2,\ XS_2,\ (l_2,\ p_8)\cdot RS'_1)\end{array}} \tag{5.30}$$

Comment: (Rule 5.30) As in the monomorphic case, the expression E is evaluated to form the body of the constructor. For example, with the polymorphic unary constructor *c_cons*, the expression would evaluate to a tuple containing a list element and the remainder of the list. The type of the constructor is derived from an *instance*-style operation. However, note that the resulting type closure is built using the result type p_7 of the constructor function, rather than the constructor type p_4.

$$\frac{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ E) \Rightarrow (H_2,\ \Delta_2,\ XS_2,\ (l_1,\ p_1) \cdot RS'_1) \\ H_2(\Delta_2(con)) = p_2 \rightarrow p_3 \qquad H_2(l_1) = con(l_2)\end{array}}{(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ \mathbf{decon}\ (con,\ E)) \Rightarrow (H_2,\ \Delta_2,\ XS_2,\ (l_2,\ p_2) \cdot RS'_1)} \tag{5.31}$$

$$\frac{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ E) \Rightarrow (H_2,\ \Delta_2,\ XS_2,\ (l_1,\ p_1) \cdot RS'_1) \\ (H_2,\ \Delta_2,\ XS_2,\ (l_1,\ p_1) \cdot RS'_1,\ \tau\_alloc(\overline{\tau}^k)) \Rightarrow \\ \quad (H_3,\ \Delta_2,\ XS_2,\ p_2 \cdot (l_1,\ p_1) \cdot RS'_1) \\ H_3(\Delta(con)) = \langle\!\langle \{\alpha^1 \mapsto p_3^1,\ \ldots,\ \alpha^k \mapsto p_3^k\},\ p_4 \rangle\!\rangle \qquad H_3(p_2) = \overline{p_5}^k \\ H_3(p_4) = p_6 \rightarrow p_7 \qquad H_3(l_1) = con(l_2) \qquad H_3 \uparrow p_8\end{array}}{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ \mathbf{decon}\ (con,\ \overline{\tau}^k,\ E)) \Rightarrow \\ \quad (H_3[p_8 \mapsto \langle\!\langle \{\alpha^1 \mapsto p_5^1,\ \ldots,\ \alpha^k \mapsto p_5^k\},\ p_6 \rangle\!\rangle],\ \Delta_2,\ XS_2,\ (l_2,\ p_8) \cdot RS'_1)\end{array}} \tag{5.32}$$

Comment: (Rule 5.31 and 5.32) Deconstruction is the reverse of construction (compare with Rules 5.28 and 5.30). Deconstruction is only applicable to unary constructors. Note that it is now the argument type of the constructor function which forms the type of the result.

Structured Expressions

$$\frac{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ E^1) \Rightarrow (H_2,\ \Delta_2,\ XS_2,\ RS_2) \cdots \\ \quad (H_k,\ \Delta_k,\ XS_k,\ RS_k,\ E^k) \Rightarrow \\ \qquad (H_{k+1},\ \Delta_{k+1},\ XS_{k+1},\ (l_1^k,\ p_1^k) \cdots (l_1^1,\ p_1^1) \cdot RS'_1) \\ H_{k+1} \uparrow l_2 \qquad H_{k+1} \uparrow p_2\end{array}}{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ \mathbf{tuple}\ \overline{E}^k) \Rightarrow \\ \quad (H_{k+1}[l_2 \mapsto (l_1^1,\ \ldots,\ l_1^k)][p_2 \mapsto (p_1^1,\ \ldots,\ p_1^k)],\ \Delta_{k+1}, \\ \qquad XS_{k+1},\ (l_2,\ p_2) \cdot RS'_1)\end{array}} \tag{5.33}$$

Comment: (Rule 5.33) A tuple is constructed by evaluating its members $\overline{E}^k$ in left-to-right order. The resulting $(l_1^k,\ p_1^k)$ pairs are kept on the result stack RS until the last one is evaluated. A tuple is then allocated on the value heap (with a corresponding type on the type heap) to hold the results.

$$\frac{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ E) \Rightarrow (H_2,\ \Delta_2,\ XS_2,\ (l_1,\ p_1) \cdot RS'_1) \\ H_2(l_1) = \overline{l_2}^k \qquad H_2(p_1) = \overline{p_2}^k\end{array}}{(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ \mathbf{select}\ (i,\ E)) \Rightarrow (H_2,\ \Delta_2,\ XS_2,\ (l_2^i,\ p_2^i) \cdot RS'_1)} \tag{5.34}$$

$$\frac{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ E_1) \Rightarrow (H_2,\ \Delta_2,\ XS_2,\ (l_1,\ p_1)\cdot RS_1')\\ H_2(l_1) = val \qquad cmap = \{c^1\ \mapsto\ E_2^1,\ \ldots,\ c^k\ \mapsto\ E_2^k\}\\ (H_2,\ \Delta_2,\ XS_2,\ RS_1',\ \text{if } (val \in \text{Dom } cmap) \text{ then } cmap(val) \text{ else } E_3) \Rightarrow\\ \quad (H_3,\ \Delta_3,\ XS_3,\ (l,\ p)\cdot RS_1'')\end{array}}{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ \textbf{switch}\ E_1\ \textbf{case}\ (cmap,\ E_3)) \Rightarrow\\ \quad (H_3,\ \Delta_3,\ XS_3,\ (l,\ p)\cdot RS_1'')\end{array}} \tag{5.35}$$

Comment: (Rule 5.35) The condition expression E_1 is evaluated to obtain the value *val*. If this value occurs in the domain of *cmap*, then the corresponding expression E_2 is evaluated. If there is no match, then the default expression E_3 is evaluated.

$$\frac{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ \Delta_1\cdot RS_1,\ \alpha\_alloc(\sigma)) \Rightarrow (H_2,\ \Delta_2,\ XS_1,\ RS_2)\\ (H_2,\ \Delta_1,\ XS_1,\ RS_2,\ E_1) \Rightarrow (H_3,\ \Delta_2,\ XS_2,\ (l_1,\ p_1)\cdot RS_2')\\ (H_3,\ \Delta_3,\ XS_2,\ (l_1,\ p_1)\cdot RS_2',\ \sigma\_alloc(\sigma,\ p_1)) \Rightarrow\\ \quad (H_4,\ \Delta_3,\ XS_2,\ p_2\cdot(l_1,\ p_1)\cdot\Delta_1'\cdot RS_1')\\ (H_4,\ \Delta_1'[x\ \mapsto\ (l_1,\ p_2)],\ XS_2,\ \Delta_1'\cdot RS_1',\ E_2) \Rightarrow\\ \quad (H_5,\ \Delta_4,\ XS_3,\ (l_2,\ p_3)\cdot\Delta_1''\cdot RS_1'')\end{array}}{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ \textbf{let}\ (x,\ \sigma)\ =\ E_1\ \textbf{in}\ E_2) \Rightarrow\\ \quad (H_5,\ \Delta_1'',\ XS_3,\ (l_2,\ p_3)\cdot RS_1'')\end{array}} \tag{5.36}$$

Comment: (Rule 5.36) The basic let rule is used to bind a single value to a variable. The complexity of this rule arises from the fact that expressions always return monomorphic types, although we may have polymorphic values. In order to remain consistent with the static semantics (compare with Rule 3.41) the evaluation is performed as follows: A copy of the environment is stored on the result stack and the type variables (if any) are allocated. The expression E_1 is evaluated to obtain the value and type $(l_1,\ p_1)$ of the variable. The type of the value is then allocated on the heap, using the σ_alloc function applied to the static type σ and the monomorphic type referenced by p_1. The old environment Δ_1' is then restored from the result stack (although a copy remains on the result stack) to remove the free type variables, as they are now bound in the type referenced by p_2. A binding is created in the main environment from the variable to its value and type $x\ \mapsto\ (l_1,\ p_2)$, and the expression E_2 is evaluated to obtain the result $(l_2,\ p_3)$. In the conclusion of the rule, the original environment Δ_1'' is again restored from the result stack to remove the variable from the current scope. The result stack is left containing only the pair $(l_2,\ p_3)$. An example of this rule in action is presented later in this chapter (Figure 5.9).

$$\frac{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ \Delta_1 \cdot RS_1,\ \alpha\_alloc(\sigma^1)) \Rightarrow (H_2,\ \Delta_2,\ XS_1,\ RS_2)\ \cdots \\ \quad (H_k,\ \Delta_k,\ XS_1,\ RS_2,\ \alpha\_alloc(\sigma^k)) \Rightarrow (H_{k+1},\ \Delta_{k+1},\ XS_1,\ RS_2) \\ (H_{k+1},\ \Delta_{k+1},\ XS_1,\ RS_2,\ E_1) \Rightarrow \\ \quad (H_{k+2},\ \Delta_{k+2},\ XS_2,\ (l_1,\ p_1) \cdot RS'_2) \\ H_{k+2}(p_1) = \overline{p_2}^{\,k} \\ (H_{k+2},\ \Delta_{k+2},\ XS_2,\ (l_1,\ p_1) \cdot RS'_2,\ \sigma\_alloc(\sigma^1,\ p_2^1)) \Rightarrow \\ \quad (H_{k+3},\ \Delta_{k+3},\ XS_2,\ RS_3) \cdots \\ \quad\quad (H_{2k+2},\ \Delta_{2k+2},\ XS_2,\ RS_k,\ \sigma\_alloc(\sigma^k,\ p_2^k)) \Rightarrow \\ \quad\quad\quad (H_{2k+3},\ \Delta_{2k+3},\ XS_2,\ p_3^k \cdots p_3^1 \cdot (l'_1,\ p'_1) \cdot \Delta'_1 \cdot RS'_1) \\ H_{k+2}(l_1) = \overline{l_2}^{\,k} \\ (H_{2k+3},\ \Delta'_1[x^1\ \mapsto\ (l_2^1,\ p_3^1),\ \ldots,\ x^k\ \mapsto\ (l_2^k,\ p_3^k)],\ XS_2,\ \Delta'_1 \cdot RS'_1,\ E_2) \Rightarrow \\ \quad (H_{2k+4},\ \Delta_{2k+4},\ XS_3,\ (l_3,\ p_4) \cdot \Delta''_1 \cdot RS''_1)\end{array}}{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ RS,\ \mathbf{let}\ \overline{(x,\ \sigma)}^{\,k}\ =\ E_1\ \mathbf{in}\ E_2) \Rightarrow \\ \quad (H_{2k+4},\ \Delta''_1,\ XS_3,\ (l_3,\ p_4) \cdot RS''_1)\end{array}} \tag{5.37}$$

Comment: (Rule 5.37) The second form of let rule binds a sequence of variables to a tuple of values. The steps performed by this rule are essentially the same as the basic let expression, though each is repeated k times.

Function Expressions

$$\frac{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ E_1) \Rightarrow (H_2,\ \Delta_2,\ XS_2,\ (l_1,\ p_1) \cdot RS'_1) \\ (H_2,\ \Delta_2,\ XS_2,\ (l_1,\ p_1) \cdot RS'_1,\ E_2) \Rightarrow (H_3,\ \Delta_3,\ XS_3,\ (l_2,\ p_2) \cdot (l'_1,\ p'_1) \cdot RS''_1) \\ H_3(l'_1) = \langle\!\langle \Delta_c,\ (x),\ E_c \rangle\!\rangle \\ (H_3,\ \Delta_c[x\ \mapsto\ (l_2,\ p_2)],\ XS_3,\ \Delta_3 \cdot RS''_1,\ E_c) \Rightarrow \\ \quad (H_4,\ \Delta_4,\ XS_4,\ (l_3,\ p_3) \cdot \Delta'_3 \cdot RS'''_1)\end{array}}{(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ \mathbf{app}\ (E_1,\ E_2)) \Rightarrow (H_4,\ \Delta'_3,\ XS_4,\ (l_3,\ p_3) \cdot RS'''_1)} \tag{5.38}$$

Comment: (Rule 5.38) The function application rule applies the function expression E_1 (which evaluates to a closure) to the argument expression E_2. Evaluation in $\mathcal{M}\Lambda$ is performed in a strict *call-by-value* manner. Firstly, both expressions are evaluated in left-to-right order. The closure is then obtained from the result of E_1, and the result of E_2 is bound to the variable x in the closure environment Δ_c. The body of the closure E_c is then evaluated in this environment. The previous environment Δ'_3 is then restored. The result of the function application remains on the result stack.

$$\frac{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ E_1) \Rightarrow (H_2,\ \Delta_2,\ XS_2,\ (l_1,\ p_1)\cdot RS'_1)\\ (H_2,\ \Delta_2,\ XS_2,\ (l_1,\ p_1)\cdot RS'_1,\ E_2^1) \Rightarrow (H_3,\ \Delta_3,\ XS_3,\ RS_2)\ \cdots\\ \quad (H_{k+1},\ \Delta_{k+1},\ XS_{k+1},\ RS_k,\ E_2^k) \Rightarrow\\ \qquad (H_{k+2},\ \Delta_{k+2},\ XS_{k+2},\ (l_2^k,\ p_2^k)\cdot(l_2^1,\ p_2^1)\cdot(l'_1,\ p'_1)\cdot RS''_1)\\ H_{k+2}(l'_1) = \langle\!\langle \Delta_c,\ \overline{x}^{\,k},\ E_c \rangle\!\rangle\\ (H_{k+2},\ \Delta_c[x^1\ \mapsto\ (l_2^1,\ p_2^1),\ \ldots,\ x^k\ \mapsto\ (l_2^k,\ p_2^k)],\ XS_{k+2},\ \Delta_{k+2}\cdot RS''_1,\ E_c) \Rightarrow\\ \quad (H_{k+3},\ \Delta_{k+3},\ XS_{k+3},\ (l_3,\ p_3)\cdot \Delta'_{k+2}\cdot RS'''_1)\end{array}}{(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ \mathbf{app}\ (E_1,\ \overline{E_2}^{\,k})) \Rightarrow (H_{k+3},\ \Delta'_{k+2},\ XS_{k+3},\ RS'''_1)} \tag{5.39}$$

Comment: (Rule 5.39) When applying a function to a sequence of arguments in a strict manner, the arguments must all be evaluated in left-to-right order before the actual application may occur.

$$\frac{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ \Delta_1\cdot RS_1,\ \alpha\_alloc(\sigma^1)) \Rightarrow (H_2,\ \Delta_2,\ XS_1,\ RS_2)\cdots\\ \quad (H_k,\ \Delta_k,\ XS_1,\ RS_2,\ \alpha\_alloc(\sigma^k)) \Rightarrow (H_{k+1},\ \Delta_{k+1},\ XS_1,\ RS_2)\\ (H_{k+1},\ \Delta_{k+1},\ XS_1,\ RS_2,\ \tau\_alloc(mono(\sigma^1))) \Rightarrow\\ \quad (H_{k+2},\ \Delta_{k+1},\ XS_1,\ RS_3)\cdots\\ \qquad (H_{2k},\ \Delta_{k+1},\ XS_1,\ RS_{k+1},\ \tau\_alloc(mono(\sigma^k))) \Rightarrow\\ \qquad\quad (H_{2k+1},\ \Delta_{k+1},\ XS_1,\ p_1^k\cdots p_1^1\cdot RS_2)\\ H_{2k+1}\uparrow (l_1^1,\ \ldots,\ l_1^k)\\ (H_{2k+1}[l_1^1\ \mapsto\ \Omega,\ \ldots,\ l_1^k\ \mapsto\ \Omega],\\ \quad \Delta_{k+1}[x^1\ \mapsto\ (l_1^1,\ p_1^1),\ \ldots,\ x^k\ \mapsto\ (l_1^k,\ p_1^k)],\ XS_1,\ RS_2,\ E_1^1) \Rightarrow\\ \qquad (H_{2k+2},\ \Delta_{k+2},\ XS_2,\ RS_3)\cdots\\ \qquad\quad (H_{3k},\ \Delta_{2k},\ XS_k,\ RS_{k+3},\ E_1^k) \Rightarrow\\ \qquad\qquad (H_{3k+1},\ \Delta_{2k+1},\ XS_{k+1},\ (l_2^k,\ p_2^k)\cdots(l_2^1,\ p_2^1)\cdot RS'_2)\\ (H_{3k+1},\ \Delta_{2k+1},\ XS_{k+1},\ RS'_2,\ \sigma\_alloc(\sigma_1,\ p_2^1)) \Rightarrow\\ \quad (H_{3k+2},\ \Delta_{2k+1},\ XS_{k+1},\ RS_3)\cdots\\ \qquad (H_{4k},\ \Delta_{2k+1},\ XS_{k+1},\ \sigma\_alloc(\sigma_1,\ p_2^1)) \Rightarrow\\ \qquad\quad (H_{4k+1},\ \Delta_{2k+1},\ XS_{k+1},\ p_3^k\cdots p_3^1\cdot \Delta'_1\cdot RS'_1)\\ (H_{4k+1}[l_1^1\ \overset{upd}{\mapsto}\ l_2^1,\ \ldots,\ l_1^k\ \overset{upd}{\mapsto}\ l_2^k],\\ \quad \Delta'_1[x^1\ \mapsto\ (l_1^1,\ p_3^1),\ \ldots,\ x^k\ \mapsto\ (l_1^k,\ p_3^k)],\ XS_{k+1},\ RS'_1,\ E_2) \Rightarrow\\ \qquad (H_{4k+2},\ \Delta_2,\ XS_{k+2},\ (l_4,\ p_4)\cdot RS''_1)\end{array}}{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ \mathbf{fix}\ \overline{(x,\ \sigma)\ =\ E_1}^{\,k}\ \mathbf{in}\ E_2) \Rightarrow\\ \quad (H_{4k+2},\ \Delta_2,\ XS_{k+2},\ (l_4,\ p_4)\cdot RS''_1)\end{array}} \tag{5.40}$$

Comment: (Rule 5.40) The fixed point rule is clearly the most complex in the abstract machine. As with the let rule, the majority of this complexity arises from the treatment of polymorphism in $\mathcal{M}\Lambda$, and the fact that we must remain consistent with the static semantics (compare with rule 3.44). Recall, that for typing purposes, the variables are bound to monomorphic types ($mono(\sigma)$) during the evaluation of the function expressions E_1, and are rebound to polymorphic types σ during the evaluation of the main expression E_2. Also, the variables must all be present in the environment before any of the function expressions can be evaluated, since any function may refer to any other. Since we do not allow dangling pointers (as they would be invalidated by garbage collection), the variables are initially bound to dummy closures Ω, and are later updated to the actual function closures.

References and Exceptions

$$\frac{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ E_1) \Rightarrow (H_2,\ \Delta_2,\ XS_2,\ (l_1,\ p_1)\cdot RS'_1)\\ (H_2,\ \Delta_2,\ XS_2,\ (l_1,\ p_1)\cdot RS'_1,\ E_2) \Rightarrow (H_3,\ \Delta_3,\ XS_3,\ (l_2,\ p_2)\cdot(l'_1,\ p'_1)\cdot RS''_1)\end{array}}{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ \textbf{assign}\ (E_1,\ E_2)) \Rightarrow\\ \quad (H_3[l'_1 \overset{upd}{\mapsto} \text{c\_ref}(l_2)],\ \Delta_3,\ XS_3,\ \Delta_3(UNIT)\cdot RS''_1)\end{array}} \tag{5.41}$$

Comment: (Rule 5.41) Assignment uses the update operation $l_1 \overset{upd}{\mapsto}$ c_ref(l_2) to update the reference at l_1 to c_ref(l_2). For consistency we return UNIT on the result stack.

$$\frac{(H_1,\ \Delta_1,\ (),\ RS_1,\ E) \Rightarrow (H_2,\ \Delta_2,\ (),\ (l,\ p)\cdot RS'_1)}{(H_1,\ \Delta_1,\ (),\ RS_1,\ \textbf{raise}\ (E,\ \tau)) \Rightarrow \textbf{halt}\ (H_2,\ \Delta_2,\ (),\ (l,\ p)\cdot RS'_1)} \tag{5.42}$$

Comment: (Rule 5.42) If there are no closures on the exception stack then a raised exception will not be handled. The effect of an un-handled exception is to halt the evaluation of the abstract machine.

$$\frac{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ E) \Rightarrow (H_2,\ \Delta_2,\ (l_1,\ p_1)\cdot XS_2,\ (l_2,\ p_2)\cdot RS'_1)\\ H_2(l_1) = \langle\!\langle \Delta_c,\ (x),\ E_c \rangle\!\rangle\\ (H_2,\ \Delta_c[x \mapsto (l_2,\ p_2)],\ XS_2,\ \Delta_2\cdot RS'_1,\ E_c) \Rightarrow\\ \quad (H_3,\ \Delta_3,\ XS_3,\ (l_3,\ p_3)\cdot\Delta'_2\cdot RS''_1)\end{array}}{(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ \textbf{raise}\ (E,\ \tau)) \Rightarrow (H_3,\ \Delta'_2,\ XS_3,\ (l_3,\ p_3)\cdot RS''_1)} \tag{5.43}$$

Comment: (Rule 5.43) If an exception is raised, and the exception stack is non-empty, the closure at the top of the exception stack is evaluated (see Rule 5.38).

$$\frac{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ E_2) \Rightarrow (H_2,\ \Delta_2,\ XS_2,\ (l_1,\ p_1) \cdot RS'_1) \\ (H_2,\ \Delta_2,\ (l_1,\ p_1) \cdot XS_2,\ RS'_1,\ E_1) \Rightarrow (H_3,\ \Delta_3,\ (l'_1,\ p'_1) \cdot XS'_2,\ (l_2,\ p_2) \cdot RS''_1)\end{array}}{(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ \textbf{handle}\ E_1\ \textbf{with}\ E_2) \Rightarrow (H_3,\ \Delta_3,\ XS_3,\ (l_2,\ p_2) \cdot RS''_1)} \quad (5.44)$$

Comment: (Rule 5.44) This rule ensures that an exception raised in E_1 is handled by E_2. This amounts to simply applying Rule 5.25 to E_2 and placing it on the exception stack while E_1 is evaluated. The **raise** rule performs the actual evaluation of the exception handler.

5.3 Example Evaluation

The machine transitions are unfortunately rather complex. This is mainly due to the efficient representation of polymorphism, and the need to carry the entire machine state through each rule. To illustrate the rules in action, we present an example in Figure 5.9. In this example we declare and bind a polymorphic identity function to the variable 'ident'. The example should be compared with the earlier type checking example in Chapter 3. Note that we are only evaluating the function expression here, not the entire program as before. Owing to page-width restrictions, we have used indentation to express the evaluation order. The rules are triggered in the following order: Rule 5.36, Rule 5.8, Rule 5.25, Rule 5.4, Rule 5.5, Rule 5.5, Rule 5.12, etc. The result of the evaluation is the pair $(l_2,\ p_4)$ on the result stack.

It is worth noting that the evaluation forms an inference-style tree structure. This is very similar to the *graph-reduction* style of evaluation, e.g. the G-machine [Pey87]. However, the actual evaluation path in our abstract machine is defined by a strict depth-first traversal of the tree. We cannot apply graph-reduction techniques such as out-of-order or lazy evaluation owing to side-effects and exceptions in our language.

It is also the case that the runtime types generated at each stage of the evaluation correspond directly to the static types generated at each stage of elaboration. In an untyped dynamic semantics, a type-soundness proof [WF94] is often performed to ensure that the static and dynamic semantics are consistent. However, this kind of proof is significantly simplified by the presence of runtime type information. In $\mathcal{M}\Lambda$ there is a one-to-one correspondence between the static and dynamic types, and so we consider this kind of proof unnecessary.

$$\left.\begin{array}{l}
\left.\dfrac{H_1 \uparrow (p_1)}{(H_1,\ \Delta_1,\ XS_1,\ \Delta_1 \cdot RS_1,\ \alpha\_\mathit{alloc}(\forall(\alpha).\,\alpha \to \alpha)) \Rightarrow (\underbrace{H_1[p_1\ \mapsto\ \alpha]}_{H_2},\ \underbrace{\Delta_1[\alpha\ \mapsto\ p_1]}_{\Delta_2},\ XS_1,\ RS_2)}\right\}(5.8)\\[2ex]
\left.\begin{array}{l}
\left.\begin{array}{l}
\left.\dfrac{}{(H_2,\ \Delta_2,\ XS_1,\ RS_2,\ \tau\_\mathit{alloc}(\alpha)) \Rightarrow (H_2,\ \Delta_2,\ XS_1,\ \underbrace{\Delta_2(\alpha)}_{=\,p_1} \cdot RS_2)}\right\}(5.5)\\[2ex]
\left.\dfrac{}{(H_2,\ \Delta_2,\ XS_1,\ p_1 \cdot RS_2,\ \tau\_\mathit{alloc}(\alpha)) \Rightarrow (H_2,\ \Delta_2,\ XS_1,\ \underbrace{\Delta_2(\alpha)}_{=\,p_1} \cdot p_1 \cdot RS_2)}\right\}(5.5)\\[2ex]
\dfrac{H_2 \uparrow p_2}{(H_2,\ \Delta_2,\ XS_1,\ RS_2,\ \tau\_\mathit{alloc}(\alpha \to \alpha)) \Rightarrow (\underbrace{H_2[p_2\ \mapsto\ p_1 \to p_1]}_{H_3},\ \Delta_2,\ XS_1,\ p_2 \cdot RS_2)}
\end{array}\right\}(5.4)\\[2ex]
\dfrac{H_3 \uparrow l_1}{(H_2,\ \Delta_2,\ XS_1,\ RS_2,\ \mathbf{fn}\ (x,\ \alpha \to \alpha)\ =\ \mathbf{var}\ x) \Rightarrow (\underbrace{H_3[l_1\ \mapsto\ \langle\!\langle \Delta,\ (x),\ \mathbf{var}\ x \rangle\!\rangle]}_{H_4},\ \Delta_2,\ XS_1,\ (l_1,\ p_2) \cdot RS_2')}
\end{array}\right\}(5.25)\\[2ex]
\left.\dfrac{H_4 \uparrow p_3}{(H_4,\ \Delta_2,\ XS_1,\ (l_1,\ p_2) \cdot RS_2',\ \sigma\_\mathit{alloc}(\forall(\alpha).\,\alpha \to \alpha,\ p_2)) \Rightarrow (\underbrace{H_4[p_3\ \mapsto\ \langle\!\langle \alpha\ \mapsto\ \Delta_2(\alpha),\ p_2 \rangle\!\rangle]}_{H_5},\ \Delta_2,\ XS_1,\ p_3 \cdot (l_1,\ p_2) \cdot \underbrace{\Delta_1' \cdot RS_1'}_{\text{from } RS_2'})}\right\}(5.12)\\[2ex]
\left.\dfrac{\ldots}{(H_5,\ \Delta_1'[\text{ident}\ \mapsto\ (l_1,\ p_3)],\ XS_1,\ \Delta_1' \cdot RS_1',\ \ldots) \Rightarrow (H_6,\ \Delta_3,\ XS_2,\ (l_2,\ p_4) \cdot \Delta_1'' \cdot RS_1'')}\right\}(\cdots)\\[2ex]
\hline
(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ \mathbf{let}\ (\text{ident},\ \forall(\alpha).\,\alpha \to \alpha)\ =\ (\mathbf{fn}\ (x,\ \alpha \to \alpha)\ =\ \mathbf{var}\ x)\ \mathbf{in}\ \ldots) \Rightarrow (H_6,\ \Delta_1'',\ XS_2,\ (l_2,\ p_4) \cdot RS_1'')
\end{array}\right\}(5.36)$$

Figure 5.9: *Evaluation Example.*

Chapter 6

Memory Management in $\mathcal{M}\Lambda$

Memory is typically modelled using a *stack* and a *heap*. The stack holds temporary values of known size whose lifetime is determined by function applications, e.g. function parameters and local variables. The heap holds all other values, e.g. closures and dynamic data structures. Memory management of the stack is relatively straightforward as values are simply added or removed from the top of the stack. By contrast, the management of the heap is considerably more challenging, and therefore forms the topic of this chapter.

Without a system for automatic heap management, the programmer is left to manage the heap using explicit allocation and deallocation facilities, e.g. `malloc` and `free` in C. For non-trivial programs this can be a very significant burden as it is, in general, very difficult to ensure that an area of memory will not be required by a later computation. If memory is deallocated too hastily, then the program will fail when it requires a value that has been removed. Conversely, if memory is deallocated too conservatively, then the program may exhaust the supply of memory available.

The prevailing technique for automatic memory management of the heap is *garbage collection*. Heap allocation is performed (implicitly) by the programmer, and deallocation by the garbage collector. Garbage collection is based on the idea that if a value is *reachable*, either from the stack or from a set of roots, then it must not be discarded. A survey of different garbage collection techniques is presented in [Wil92] and [JL96]. In this chapter we present an abstract machine formalism of garbage collection which integrates with the dynamic semantics of $\mathcal{M}\Lambda$. Our formalisation is based on a two-space copying garbage collection algorithm which utilises the *tag-free* technique described in [Tol94]. We will extend this formalism to a distributed setting in Chapter 8. As with the dynamic semantics, the distributed case is a generalisation of the sequential case. Earlier versions of this abstract machine appeared in [Wal99] and [WKG00].

6.1 Copying Garbage Collection

Before proceeding with our definition, we will briefly describe the basic two-space copying algorithm, illustrated in Figure 6.1. The advantages of the two-space technique are well known, for example, the data is compacted during collection which improves locality, and cyclic garbage is removed. The address space of the heap is divided into two contiguous *semi-spaces*. During normal program execution, only one of these semi-spaces is used. Memory is allocated in a linear fashion until garbage collection appears to be profitable. At this point, the copying collector is called to reclaim space. The current semi-space (*from* space) is recursively scanned from the root objects in a depth-first fashion, and all reachable objects (via pointers) are copied into the other semi-space (*to* space). When all of the objects that are reachable from the roots have been copied, the collection is finished, and the old semi-space (*from* space) can be discarded. Subsequent memory allocations are performed in the new space (*to* space). The role of the semi-spaces is then reversed for the next garbage collection.

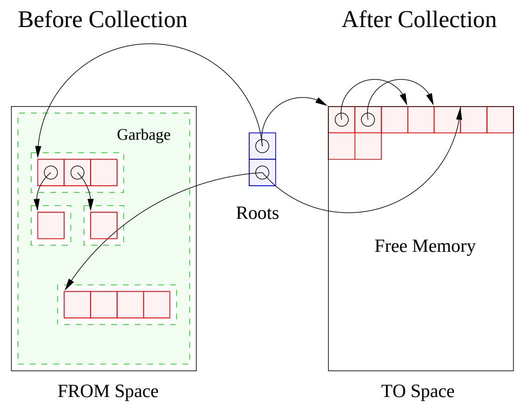


Figure 6.1: *Two-Space Copying Garbage Collection*

6.1.1 Tag-free Collection

One of the main difficulties with garbage collection is distinguishing between pointers and data values during the scanning process. For example, it is not immediately apparent whether location 102 of the value heap in Figure 6.2 contains the integer 100 or a pointer to location 100. Therefore, garbage-collected languages typically use headers and extra data fields, called *tags*, to store the lengths of data items, and to distinguish between pointers and integers. Removing these tags would considerably decrease the space requirements of a program, and save allocation time. Furthermore, it would allow a more natural style of representation, much closer that used by the machine, and therefore save processing time, e.g. removing the tags on integers when performing arithmetic operations. This approach was first demonstrated in the STG machine [Pey92].

Our garbage collection algorithm adopts the typed tag-free style of [Tol94]. In principle, this technique may be applied to any language with strong static typing. In Chapter 5, we paired every location l with a pointer p. We can use the type information referenced by p to determine the *shape* of the value referenced by l, and therefore distinguish between pointers and values without resorting to the use of tags. This gives rise to a symmetric garbage collection; during the collection process the collector continually maintains one pointer in the value heap, and another pointer on the corresponding type in the type heap. This is illustrated in Figure 6.2. The value 1 at location 100 in the value heap is paired with the type t_bool at location 12 in the type heap. The garbage collector can therefore determine that the value 1 corresponds to the boolean truth value and is not a pointer. The value 102 at location 101 in the value heap is paired with the type t_ref(11). Since references are represented by pointers, the garbage collector determines that the value 102 is a pointer, and the corresponding type is at location 11 in the type heap. Thus, the value 100 at location 102 is paired with the type t_int and is therefore an integer. Note that we also garbage collect the type information as a separate operation, as we may have types that are no longer required.

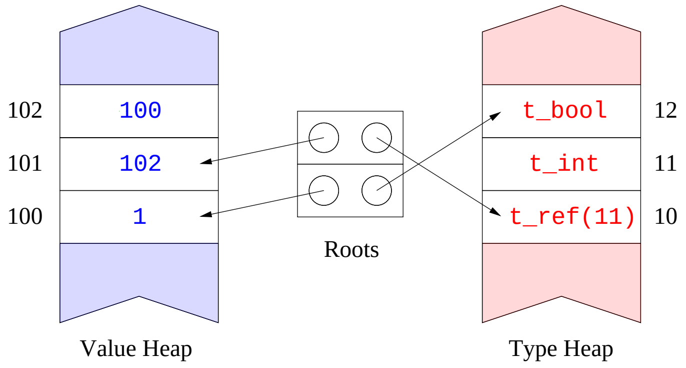


Figure 6.2: *Tag-free Collection*

It may be argued that we have not really eliminated the tags, rather we have simply moved them into the type heap. However, we now only require one type t_int in the type heap for all of the integers in the value heap, potentially a considerable saving in space. Furthermore, we now have detailed information on the values instead of simple tags. As we will see in later chapters, this information is also of use for code replacement and optimisation.

6.2 Abstract Machine Garbage Collection

We define garbage collection using an abstract machine model. The syntax of the abstract machine is defined in Figure 6.3. The state of the garbage collector is the 4-tuple (Hf, Ht, PF, LF). The *from* semi-space (heap) is denoted Hf, and the *to* semi-space (heap) is denoted Ht. When a type or a value is copied from Hf into Ht, an entry is created in either PF or LF respectively. An entry $p_1 \mapsto p_2$ in PF represents a copied type where p_1 is the old type-pointer in Hf, and p_2 is the new type-pointer in Ht. Similarly, an entry $l_1 \mapsto l_2$ in LF represents a copied value. These forwarding tables are used during garbage collection to ensure that a type or value is copied only once between spaces. The type environment TE is required for garbage collecting values with polymorphic types, due to the deferred instantiation method described in Chapter 5.

Garbage Collection State	GC	$::=$	(Hf, Ht, PF, LF, TE)
From Semi-space	Hf	$::=$	H
To Semi-space	Ht	$::=$	H
Forwarding Type Pointer	PF	$::=$	$p_1 \mapsto p_2$
Forwarding Value Pointer	LF	$::=$	$l_1 \mapsto l_2$
Type Environment	TE	$::=$	$\alpha \overset{map}{\mapsto} p$

Figure 6.3: *Abstract Machine Syntax.*

6.2.1 Top-level Machine Transitions

Garbage collection is defined by a transition between states of the abstract machine. Garbage collection is incorporated into the dynamic semantics of $\mathcal{M}\Lambda$ through the Rule 6.1. The roots for the collection are the environment Δ, the exception stack XS, and the result stack RS, at the point when garbage collection is initiated. We define the rules for collection of these roots below. The complex issue of when to initiate a collection will not be dealt with in detail here. However, a discussion can be found in [App92] where the author arrives at the following rule:

- For some constant $R > 1$, when current memory usage is more than R times the amount of reachable data preserved by the previous garbage collection, start a new garbage collection.

Initiation

$$\frac{\begin{array}{l}(Hf,\ \emptyset,\ \emptyset,\ \emptyset,\ \emptyset,\ \Delta_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_1,\ PF_1,\ LF_1,\ TE,\ \Delta_2)\\ (Hf,\ Ht_1,\ PF_1,\ LF_1,\ TE,\ XS_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF_2,\ TE,\ XS_2)\\ (Hf,\ Ht_2,\ PF_2,\ LF_2,\ TE,\ RS_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_3,\ PF_3,\ LF_3,\ TE,\ RS_2)\\ (Ht_3,\ \Delta_2,\ XS_2,\ RS_2,\ E) \Rightarrow (Ht_4,\ \Delta_3,\ XS_3,\ RS_3)\end{array}}{(Hf,\ \Delta_1,\ XS_1,\ RS_1,\ E) \Rightarrow (Ht_4,\ \Delta_3,\ XS_3,\ RS_3)} \tag{6.1}$$

Comment: (Rule 6.1) Collection can occur when an expression (or sub-expression) E is to be evaluated (using the rules from Chapter 5). At this point, the machine is interrupted, the garbage collector is initialised, and collection proceeds from each of the roots in turn. Once collection has completed, Hf is discarded, and the evaluation of E resumes with the new abstract machine state.

Stack Collection

$$\frac{}{(Hf,\ Ht,\ PF,\ LF,\ TE,\ ()) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht,\ PF,\ LF,\ TE,\ ())} \tag{6.2}$$

$$\frac{\begin{array}{l}(Hf,\ Ht_1,\ PF_1,\ LF_1,\ TE,\ p_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF_1,\ TE,\ p_2)\\ (Hf,\ Ht_2,\ PF_2,\ LF_1,\ TE,\ S_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_3,\ PF_3,\ LF_2,\ TE,\ S_2)\end{array}}{(Hf,\ Ht_1,\ PF_1,\ LF_1,\ TE,\ p_1 \cdot S_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_3,\ PF_3,\ LF_2,\ TE,\ p_2 \cdot S_2)} \tag{6.3}$$

$$\frac{\begin{array}{l}(Hf,\ Ht_1,\ PF_1,\ LF_1,\ TE,\ p_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF_1,\ TE,\ p_2)\\ (Hf,\ Ht_2,\ PF_2,\ LF_1,\ TE,\ (l_1,\ p_1)) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_3,\ PF_3,\ LF_2,\ TE,\ (l_2,\ p_2))\\ (Hf,\ Ht_3,\ PF_3,\ LF_2,\ TE,\ S_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_4,\ PF_4,\ LF_3,\ TE,\ S_2)\end{array}}{\begin{array}{l}(Hf,\ Ht_1,\ PF_1,\ LF_1,\ TE,\ (l_1,\ p_1) \cdot S_1) \Rightarrow_{\mathrm{gc}}\\ \quad (Hf,\ Ht_4,\ PF_4,\ LF_3,\ TE,\ (l_2,\ p_2) \cdot S_2)\end{array}} \tag{6.4}$$

$$\frac{\begin{array}{l}(Hf,\ Ht_1,\ PF_1,\ LF_1,\ TE,\ \Delta_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF_2,\ TE,\ \Delta_2)\\ (Hf,\ Ht_2,\ PF_2,\ LF_2,\ TE,\ S_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_3,\ PF_3,\ LF_3,\ TE,\ S_2)\end{array}}{(Hf,\ Ht_1,\ PF_1,\ LF_1,\ TE,\ \Delta_1 \cdot S_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_3,\ PF_3,\ LF_3,\ TE,\ \Delta_2 \cdot S_2)} \tag{6.5}$$

Comment: (Rules 6.2 to 6.5) The garbage collection of the stacks XS and RS is defined inductively. The base case is Rule 6.2; an empty stack requires no collection. The remaining rules perform the recursive step; the item on the top of the stack is garbage collected to give a new item, then the remainder of the stack S_1 is garbage collected to give a new stack S_2, finally the item is pushed onto the top of the stack S_2. The stacks may contain type pointers (Rule 6.3), pairs of locations and pointers (Rule 6.4), and environments (Rule 6.5).

Environment Collection

$$\frac{\begin{array}{l}
\Delta_1 = (TE_1,\ VE_1,\ DE_1,\ CE_1) \qquad TE_1 = \{\alpha^1 \mapsto p_1^1,\ \ldots,\ \alpha^k \mapsto p_1^k\} \\
(Hf,\ Ht_1,\ PF_1,\ LF_1,\ TE,\ p_1^1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF_1,\ TE,\ p_2^1)\ \cdots \\
\quad (Hf,\ Ht_k,\ PF_k,\ LF_1,\ TE,\ p_1^k) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_{k+1},\ PF_{k+1},\ LF_1,\ TE,\ p_2^k) \\
TE_2 = \{\alpha^1 \mapsto p_2^1,\ \ldots,\ \alpha^k \mapsto p_2^k\} \\
VE_1 = \{x^1 \mapsto (l_1^1,\ p_3^1),\ \ldots,\ x^l \mapsto (l_1^l,\ p_3^l)\} \\
(Hf,\ Ht_{k+1},\ PF_{k+1},\ LF_1,\ TE,\ p_3^1) \Rightarrow_{\mathrm{gc}} \\
\quad (Hf,\ Ht_{k+2},\ PF_{k+2},\ LF_1,\ TE,\ p_4^1)\ \cdots \\
\qquad (Hf,\ Ht_{k+l},\ PF_{k+l},\ LF_1,\ TE,\ p_3^l) \Rightarrow_{\mathrm{gc}} \\
\qquad\quad (Hf,\ Ht_{k+l+1},\ PF_{k+l+1},\ LF_1,\ TE,\ p_4^l) \\
(Hf,\ Ht_{k+l+1},\ PF_{k+l+1},\ LF_1,\ TE,\ (l_1^1,\ p_3^1)) \Rightarrow_{\mathrm{gc}} \\
\quad (Hf,\ Ht_{k+l+2},\ PF_{k+l+2},\ LF_2,\ TE,\ (l_2^1,\ p_4^1))\ \cdots \\
\qquad (Hf,\ Ht_{k+2l},\ PF_{k+2l},\ LF_l,\ TE,\ (l_1^l,\ p_3^l)) \Rightarrow_{\mathrm{gc}} \\
\qquad\quad (Hf,\ Ht_{k+2l+1},\ PF_{k+2l+1},\ LF_{l+1},\ TE,\ (l_2^l,\ p_4^l)) \\
VE_2 = \{x^1 \mapsto (l_2^1,\ p_4^1),\ \ldots,\ x^l \mapsto (l_2^l,\ p_4^l)\} \\
DE_1 = \{tn^1 \mapsto p_5^1,\ \ldots,\ tn^m \mapsto p_5^m\} \\
(Hf,\ Ht_{k+2l+1},\ PF_{k+2l+1},\ LF_{l+1},\ TE,\ p_5^1) \Rightarrow_{\mathrm{gc}} \\
\quad (Hf,\ Ht_{k+2l+2},\ PF_{k+2l+2},\ LF_{l+1},\ TE,\ p_6^1)\ \cdots \\
\qquad (Hf,\ Ht_{k+2l+m},\ PF_{k+2l+m},\ LF_{l+1},\ TE,\ p_5^m) \Rightarrow_{\mathrm{gc}} \\
\qquad\quad (Hf,\ Ht_{k+2l+m+1},\ PF_{k+2l+m+1},\ LF_{l+1},\ TE,\ p_6^m) \\
DE_2 = \{tn^1 \mapsto p_6^1,\ \ldots,\ tn^m \mapsto p_6^m\} \\
CE_1 = \{con^1 \mapsto p_7^1,\ \ldots,\ con^n \mapsto p_7^n\} \\
(Hf,\ Ht_{k+2l+m+1},\ PF_{k+2l+m+1},\ LF_{l+1},\ TE,\ p_7^1) \Rightarrow_{\mathrm{gc}} \\
\quad (Hf,\ Ht_{k+2l+m+2},\ PF_{k+2l+m+2},\ LF_{l+1},\ TE,\ p_8^1)\ \cdots \\
\qquad (Hf,\ Ht_{k+2l+m+n},\ PF_{k+2l+m+n},\ LF_{l+1},\ TE,\ p_7^n) \Rightarrow_{\mathrm{gc}} \\
\qquad\quad (Hf,\ Ht_{k+2l+m+n+1},\ PF_{k+2l+m+n+1},\ LF_{l+1},\ TE,\ p_8^n) \\
CE_2 = \{con^1 \mapsto p_8^1,\ \ldots,\ con^n \mapsto p_8^n\} \qquad \Delta_2 = (TE_2,\ VE_2,\ DE_2,\ CE_2)
\end{array}}{\begin{array}{l}
(Hf,\ Ht_1,\ PF_1,\ LF_1,\ TE,\ \Delta_1) \Rightarrow_{\mathrm{gc}} \\
\quad (Hf,\ Ht_{k+2l+m+n+1},\ PF_{k+2l+m+n+1},\ LF_{l+1},\ TE,\ \Delta_2)
\end{array}} \qquad (6.6)$$

Comment: (Rule 6.6) Garbage collecting an environment Δ_1 yields a new environment Δ_2. The environment Δ_1 is decomposed into sub-environments TE_1, VE_1, DE_1, and CE_1. These sub-environments are further decomposed into sequences of mappings. All of the pointers and locations contained within the ranges of these mappings are garbage collected, and new mappings are constructed from the results. The old mappings reference the *from* heap Hf, while the new mapping reference the *to* heap Ht. These new mappings are subsequently combined into new sub-environments, resulting in the new environment Δ_2.

6.2.2 Type Collection

We define the collection of the type information in this section. Unlike the collection of the values which follows, there is no potential confusion between integers and pointers, and therefore we can perform a straightforward copying collection. In actual fact, a copy of the types is not performed, rather the reachable types are reallocated from scratch in the *to* heap. This reallocation process effects the compaction of the type information.

Garbage collection of the types is defined by Rule 6.7 through 6.13. Each of the rules attempts the collection of a type, referenced by a pointer p_1, in the *from* heap Hf. Each of these types is essentially a graph structure in Hf. Collection corresponds to a depth-first traversal of this graph. As the collector descends the graph, a pointer is reserved in Ht to hold the type, although a copy is not made at this stage, and an entry is created in the forwarding table which maps the old pointer to the new pointer $\mathit{PF}[p_1 \mapsto p_2]$. As the collector re-ascends the graph, the types are copied (i.e. reallocated) into Ht using the pointers returned from the collection of the child types. Cycles in the traversal are eliminated by Rule 6.7, which ensures that a type is collected only once. This is achieved through a comparison of the pointer p_1 with the entries in the forwarding table PF. Each of the remaining rules performs the collection of a different type. If the type is a type name (Rule 6.8), or type variable (Rule 6.12), and therefore contains no children, then the type is simply reallocated in Ht and the collection is complete. However, if the type contains pointers to child types, then the rules are invoked recursively to perform the traversal. Each rule returns a pointer to the collected type in Ht.

$$\frac{p_1 \in \mathrm{Dom}\ \mathit{PF}}{(\mathit{Hf},\ \mathit{Ht},\ \mathit{PF},\ \mathit{LF},\ \mathit{TE},\ p_1) \Rightarrow_{\mathrm{gc}} (\mathit{Hf},\ \mathit{Ht},\ \mathit{PF},\ \mathit{LF},\ \mathit{TE},\ \mathit{PF}(p_1))} \tag{6.7}$$

Comment: (Rule: 6.7) This rule ensures that we do not collect a type more than once. We simply return a pointer to the type in Ht obtained from PF.

$$\frac{p_1 \notin \mathrm{Dom}\ \mathit{PF} \qquad \mathit{Hf}(p_1) = \mathit{tn} \qquad \mathit{Ht} \uparrow p_2}{\begin{array}{l}(\mathit{Hf},\ \mathit{Ht},\ \mathit{PF},\ \mathit{LF},\ \mathit{TE},\ p_1) \Rightarrow_{\mathrm{gc}} \\ \quad (\mathit{Hf},\ \mathit{Ht}[p_2 \mapsto \mathit{tn}],\ \mathit{PF}[p_1 \mapsto p_2],\ \mathit{LF},\ \mathit{TE},\ p_2)\end{array}} \tag{6.8}$$

Comment: (Rule: 6.8) A type name (corresponding to a special constant or nullary constructor) contains no additional type pointers. Therefore, the type name is allocated directly in Ht, and the pointer p_2 returned.

$$\frac{\begin{array}{l} p_1 \notin \mathrm{Dom}\, PF_1 \qquad Hf(p_1) = tn(p_2) \qquad Ht_1 \uparrow p_3 \\ (Hf,\ Ht_1,\ PF_1[p_1 \mapsto p_3],\ LF,\ TE,\ p_2) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF,\ TE,\ p_4) \end{array}}{\begin{array}{l} (Hf,\ Ht_1,\ PF_1,\ LF,\ TE,\ p_1) \Rightarrow_{\mathrm{gc}} \\ \quad (Hf,\ Ht_3[p_3 \mapsto tn(p_4)],\ PF_3,\ LF,\ TE,\ p_3) \end{array}} \tag{6.9}$$

Comment: (Rule 6.9) A constructor type contains a single type pointer: $tn(p_2)$. The pointer p_2 is collected yielding a new pointer p_4, which is combined with the original type name to form the new constructor type $tn(p_4)$. Note that the mapping $p_1 \mapsto p_3$ is added to the forwarding table PF_1 before these types are collected.

$$\frac{\begin{array}{l} p_1 \notin \mathrm{Dom}\, PF_1 \qquad Hf(p_1) = \overline{p_2}^{k} \qquad Ht_1 \uparrow p_3 \\ (Hf,\ Ht_1,\ PF_1[p_1 \mapsto p_3],\ LF,\ TE,\ p_2^1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF,\ TE,\ p_4^1) \cdots \\ \quad (Hf,\ Ht_k,\ PF_k,\ LF,\ TE,\ p_2^k) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_{k+1},\ PF_{k+1},\ LF,\ TE,\ p_4^k) \end{array}}{\begin{array}{l} (Hf,\ Ht_1,\ PF_1,\ LF,\ TE,\ p_1) \Rightarrow_{\mathrm{gc}} \\ \quad (Hf,\ Ht_{k+1}[p_3 \mapsto \overline{p_4}^{k}],\ PF_{k+1},\ LF,\ TE,\ p_3) \end{array}} \tag{6.10}$$

$$\frac{\begin{array}{l} p_1 \notin \mathrm{Dom}\, PF_1 \qquad Hf(p_1) = p_2 \rightarrow p_3 \qquad Ht_1 \uparrow p_4 \\ (Hf,\ Ht_1,\ PF_1[p_1 \mapsto p_4],\ LF,\ TE,\ p_2) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF,\ TE,\ p_5) \\ (Hf,\ Ht_2,\ PF_2,\ LF,\ TE,\ p_3) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_3,\ PF_3,\ LF,\ TE,\ p_6) \end{array}}{\begin{array}{l} (Hf,\ Ht_1,\ PF_1,\ LF,\ TE,\ p_1) \Rightarrow_{\mathrm{gc}} \\ \quad (Hf,\ Ht_3[p_4 \mapsto p_5 \rightarrow p_6],\ PF_3,\ LF,\ TE,\ p_4) \end{array}} \tag{6.11}$$

$$\frac{p_1 \notin \mathrm{Dom}\, PF \qquad Hf(p_1) = \alpha \qquad Ht \uparrow p_2}{\begin{array}{l} (Hf,\ Ht,\ PF,\ LF,\ TE,\ p_1) \Rightarrow_{\mathrm{gc}} \\ \quad (Hf,\ Ht[p_2 \mapsto \alpha],\ PF[p_1 \mapsto p_2],\ LF,\ TE,\ p_2) \end{array}} \tag{6.12}$$

$$\frac{\begin{array}{l} p_1 \notin \mathrm{Dom}\, PF_1 \qquad Hf(p_1) = \langle\!\langle TE_1,\ p_2 \rangle\!\rangle \qquad Ht_1 \uparrow p_3 \\ TE_1 = \{\alpha^1 \mapsto p_4^1,\ \ldots,\ \alpha^k \mapsto p_4^k\} \\ (Hf,\ Ht_1,\ PF_1[p_1 \mapsto p_3],\ LF,\ TE,\ p_4^1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF,\ TE,\ p_5^1) \cdots \\ \quad (Hf,\ Ht_k,\ PF_k,\ LF,\ TE,\ p_4^k) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_{k+1},\ PF_{k+1},\ LF,\ TE,\ p_5^k) \\ TE_2 = \{\alpha^1 \mapsto p_5^1,\ \ldots,\ \alpha^k \mapsto p_5^k\} \\ (Hf,\ Ht_{k+1},\ PF_{k+1},\ LF,\ TE,\ p_2) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_{k+2},\ PF_{k+2},\ LF,\ TE,\ p_6) \end{array}}{\begin{array}{l} (Hf,\ Ht_1,\ PF_1,\ LF,\ TE,\ p_1) \Rightarrow_{\mathrm{gc}} \\ \quad (Hf,\ Ht_{k+2}[p_3 \mapsto \langle\!\langle TE_2,\ p_6 \rangle\!\rangle],\ PF_{k+2},\ LF,\ TE,\ p_3) \end{array}} \tag{6.13}$$

Comment: (Rule: 6.13) A type closure contains a type environment. We collect this environment TE_1 in a similar manner to Rule 6.6 by decomposing the environment into mappings, collecting the ranges of these mappings, and constructing a new type environment TE_2.

6.2.3 Value Collection

Garbage collection of the values is defined in this section. As with the collection of types, the collection corresponds to a depth-first traversal of the graph represented by each value. However, we now have the potential for confusion between integers and pointers, and therefore resort to the symmetric tag-free approach to collection described earlier; at each stage of the collection we use the type to determine the shape of the value.

The collection of the values is defined by Rule 6.14 through Rule 6.21. Each rule operates on a pair $(l_1,\ p_1)$ which references a value and the corresponding type in Hf. Rule 6.14 ensures that we do not collect a value more than once by comparing l_1 with the forwarding table LF. Each of the remaining rules performs the collection of a different value. The premises of these rules are very similar. We first check that the value has not already been copied $l_1 \notin \text{Dom } LF_1$. We then examine the type $\mathit{Hf}(p_1)$ associated with the value. This type information is used to determine the shape of the value. Once the shape is known, we may extract the value from the heap $\mathit{Hf}(l_1)$. We then reserve a location $\mathit{Ht} \uparrow l_2$ to hold the copied value and update the forwarding table to point to this new location $LF[l_1 \ \mapsto\ l_2]$. The remaining steps depend on the value. If the value is a special constant or constructor (Rule 6.15) then we simply reallocate the value in Ht and return the new location. However, if the value contains any references to other locations, we must collect these in depth-first fashion as before. The result of each rule is a pair $(l_2,\ p_2)$ containing a reference to the collected value, and its corresponding type in Ht. The type pointer p_2 is obtained from the forwarding table PF. This pointer will always be available as the types of the lambda variables are collected before their values (Rule 6.6). This step is necessary as we must always pair a location with a pointer in $\mathcal{M}\Lambda$, and it would be undesirable to pair a location in Ht with a pointer in Hf.

$$\frac{l_1 \in \text{Dom } LF \qquad LF(l_1) = l_2}{(\mathit{Hf},\ \mathit{Ht},\ PF,\ LF,\ TE,\ (l_1,\ p_1)) \Rightarrow_{\text{gc}} (\mathit{Hf},\ \mathit{Ht},\ PF,\ LF,\ TE,\ (l_2,\ PF(p_1)))} \tag{6.14}$$

$$\frac{l_1 \notin \text{Dom } LF \quad \mathit{Hf}(p_1) = \mathit{tn} \qquad \mathit{Ht} \uparrow l_2}{\begin{array}{l}(\mathit{Hf},\ \mathit{Ht},\ PF,\ LF,\ TE,\ (l_1,\ p_1)) \Rightarrow_{\text{gc}} \\ \quad (\mathit{Hf},\ \mathit{Ht}[l_2 \ \mapsto\ \mathit{Hf}(l_1)],\ PF,\ LF[l_1 \ \mapsto\ l_2],\ TE,\ (l_2,\ PF(p_1)))\end{array}} \tag{6.15}$$

Comment: (Rule 6.15) A special constant or nullary constructor, with corresponding type tn, contains no additional references and may simply be copied between heaps $\mathit{Ht}[l_2 \ \mapsto\ \mathit{Hf}(l_1)]$.

$$\frac{\begin{array}{l} l_1 \notin \text{Dom}\ LF_1 \qquad Hf(p_1) = tn(p_2) \qquad Hf(l_1) = con(l_2) \qquad Ht_1 \uparrow l_3 \\ (Hf,\ Ht_1,\ PF_1,\ LF_1[l_1\ \mapsto\ l_3],\ TE,\ (l_2,\ p_2)) \Rightarrow_{\text{gc}} \\ \quad (Hf,\ Ht_2,\ PF_2,\ LF_2,\ TE,\ (l_4,\ p_3)) \end{array}}{\begin{array}{l} (Hf,\ Ht_1,\ PF_1,\ LF_1,\ TE,\ (l_1,\ p_1)) \Rightarrow_{\text{gc}} \\ \quad (Hf,\ Ht_2[l_3\ \mapsto\ con(l_4)],\ PF_2,\ LF_2,\ TE,\ (l_3,\ PF_2(p_1))) \end{array}} \tag{6.16}$$

Comment: (Rule 6.16) A value constructor $con(l_2)$ contains a reference to a value l_2. We must therefore copy this value before we can copy the value constructor. As this value may itself reference other values, we must invoke the garbage collection rules recursively to copy this value. We obtain the type of the value from the type of the value constructor $p_2(p_3)$. We construct a new pair $(l_2,\ p_3)$ and invoke the rules to obtain a new pair $(l_4,\ p_4)$ which reference the copied value in Ht. Finally, we copy and update the value constructor $Ht[l_4\ \mapsto\ con(l_4)]$.

$$\frac{\begin{array}{l} l_1 \notin \text{Dom}\ LF_1 \qquad Hf(p_1) = \overline{p_2}^{\,k} \qquad Hf(l_1) = \overline{l_2}^{\,k} \qquad Ht_1 \uparrow l_3 \\ (Hf,\ Ht_1,\ PF_1[l_1\ \mapsto\ l_3],\ LF_1,\ TE,\ (l_2^1,\ p_2^1)) \Rightarrow_{\text{gc}} \\ \quad (Hf,\ Ht_2,\ PF_2,\ LF_2,\ TE,\ (l_4^1,\ p_3^1))\ \cdots \\ \qquad (Hf,\ Ht_k,\ PF_k,\ LF_k,\ TE,\ (l_2^k,\ p_2^k)) \Rightarrow_{\text{gc}} \\ \qquad\quad (Hf,\ Ht_{k+1},\ PF_{k+1},\ LF_{k+1},\ TE,\ (l_4^k,\ p_3^k)) \end{array}}{\begin{array}{l} (Hf,\ Ht_1,\ PF_1,\ LF_1,\ TE,\ (l_1,\ p_1)) \Rightarrow_{\text{gc}} \\ \quad (Hf,\ Ht_{k+1}[l_3\ \mapsto\ \overline{l_4}^{\,k}],\ PF_{k+1},\ LF_{k+1},\ TE,\ (l_3,\ PF_{k+1}(p_1))) \end{array}} \tag{6.17}$$

Comment: (Rule 6.17) A tuple $\overline{l_2}^{\,k}$ references a sequence of values. We therefore adopt the same strategy as Rule 6.16 for copying these values. Using the type of the tuple $\overline{p_2}^{\,k}$, we construct and garbage collect a sequence of pairs $(l_2^1,\ p_2^1)$ to $(l_2^k,\ p_2^k)$ which yields a new sequence of pairs (l_4^1, p_3^1) to (l_4^k, p_3^k). Finally, we construct a new tuple $Ht_{k+1}[l_3\ \mapsto\ \overline{l_4}^{\,k}]$.

$$\frac{\begin{array}{l} l_1 \notin \text{Dom}\ LF_1 \qquad Hf(p_1) = p_2 \rightarrow p_3 \qquad Hf(l_1) = \langle\!\langle \Delta_1,\ \overline{x}^{\,k},\ E \rangle\!\rangle \qquad Ht_1 \uparrow l_2 \\ (Hf,\ Ht_1,\ PF_1,\ LF_1[l_1\ \mapsto\ l_2],\ TE,\ \Delta_1) \Rightarrow_{\text{gc}} \\ \quad (Hf,\ Ht_2,\ PF_2,\ LF_2,\ TE,\ \Delta_2) \end{array}}{\begin{array}{l} (Hf,\ Ht_1,\ PF_1,\ LF_1,\ TE,\ (l_1,\ p_1)) \Rightarrow_{\text{gc}} \\ \quad (Hf,\ Ht_2[l_2\ \mapsto\ \langle\!\langle \Delta_2,\ \overline{x}^{\,k},\ E \rangle\!\rangle],\ PF_2,\ LF_2,\ TE,\ (l_2,\ PF_2(p_1))) \end{array}} \tag{6.18}$$

Comment: (Rule 6.18) A closure $\langle\!\langle \Delta_1,\ \overline{x}^{\,k},\ E \rangle\!\rangle$ contains an environment Δ_1 which must be collected. We therefore invoke Rule 6.6 which yields a new environment Δ_2. Finally, we copy and update the closure $Ht[l_2\ \mapsto\ \langle\!\langle \Delta_2,\ \overline{x}^{\,k},\ E \rangle\!\rangle]$.

$$\frac{l_1 \notin \mathrm{Dom}\, LF_1 \qquad Hf(p_1) = p_2 \to p_3 \qquad Hf(l_1) = \Omega \qquad Ht \uparrow l_2}{\begin{array}{l}(Hf,\ Ht_1,\ PF_1,\ LF_1,\ TE,\ (l_1,\ p_1)) \Rightarrow_{\mathrm{gc}} \\ \quad (Hf,\ Ht_2[l_2 \mapsto \Omega],\ PF_1,\ LF_2[l_1 \mapsto l_2],\ TE,\ (l_2,\ PF_2(p_1)))\end{array}} \tag{6.19}$$

Comment: (Rule 6.19) A dummy closure contains no environment and is therefore simply copied into Ht.

$$\frac{\begin{array}{l} l_1 \notin \mathrm{Dom}\, LF_1 \qquad Hf(p_1) = \alpha \\ (Hf,\ Ht_1,\ PF_1,\ LF_1,\ TE,\ (l_1,\ TE_1(\alpha))) \Rightarrow_{\mathrm{gc}} \\ \quad (Hf,\ Ht_2,\ PF_2,\ LF_2,\ TE,\ (l_2,\ p_2))\end{array}}{\begin{array}{l}(Hf,\ Ht_1,\ PF_1,\ LF_1,\ TE,\ (l_1,\ p_1)) \Rightarrow_{\mathrm{gc}} \\ \quad (Hf,\ Ht_2,\ PF_2,\ LF_2,\ TE,\ (l_2,\ PF_2(p_1)))\end{array}} \tag{6.20}$$

$$\frac{\begin{array}{l} l_1 \notin \mathrm{Dom}\, LF_1 \qquad Hf(p_1) = \langle\!\langle TE,\ p_2 \rangle\!\rangle \\ (Hf,\ Ht_1,\ PF_1,\ LF_1,\ TE,\ (l_1,\ p_2)) \Rightarrow_{\mathrm{gc}} \\ \quad (Hf,\ Ht_2,\ PF_2,\ LF_2,\ TE,\ (l_2,\ p_2))\end{array}}{\begin{array}{l}(Hf,\ Ht_1,\ PF_1,\ LF_1,\ \emptyset,\ (l_1,\ p_1)) \Rightarrow_{\mathrm{gc}} \\ \quad (Hf,\ Ht_2,\ PF_2,\ LF_2,\ \emptyset,\ (l_2,\ PF_2(p_1)))\end{array}} \tag{6.21}$$

Comment: (Rule 6.20 and 6.21) As a result of the deferred instantiation of polymorphic types, defined in Chapter 5, a value may be paired with an instance of a polymorphic type, represented by a type closure $\langle\!\langle TE,\ p_2 \rangle\!\rangle$ (Rule 6.21). In order to collect such values, we use the type environment of the garbage collection abstract machine to store the mappings between the type variables and their types, contained within the closure. We then invoke the value collection rules recursively with the pair $(l_1,\ p_2)$ and continue as before. On completing the collection of the value, the type environment is cleared. Since $\mathcal{M}\Lambda$ does not permit type schemes to be nested, we require only one type environment in the garbage collection machine, which will always be empty at the beginning of the rule. In the subsequent depth-first traversal, we may encounter a value paired with a type variable (Rule 6.20). However, we can now obtain the necessary type from the type environment, and thus we simply continue the collection with the pair $(l_1,\ TE(\alpha))$.

6.3 Alternative Algorithms

The abstract machine model of garbage collection presented in this chapter is both novel and powerful. We have presented a formalism of a two-space copying collector which makes all the details of memory management visible at just the right level of detail for an implementation. It is worth noting that the abstract machine model may easily be used to represent other garbage collection algorithms. For example, a generational copying collector could be defined by further segmentation of the heaps. Similarly, a mark and sweep collector may be expressed by maintaining a list of free pointers and collecting over a single heap. In both cases, the tag-free technique may be used, essentially without alteration, to improve the efficiency of the collection. In Chapter 9 we define a distributed variant of our copying collector. By introducing concurrency into the model, we can further improve the efficiency of the collection. For example, we are no longer forced to artificially order the collection of the roots as in Rule 6.1, rather we can collect from all of the roots simultaneously. In this chapter we also address the verification of the garbage collection algorithm. This is an essential step as an incorrect algorithm will have a devastating effect on program execution behaviour. In particular, we verify the algorithm for progress (correct termination), preservation (no dangling pointers), and correctness (no corrupt data). The sequential algorithm of this chapter corresponds directly to the distributed algorithm executing a single thread on a single processor.

$$\frac{\begin{array}{l}\Delta_1 = (\mathit{TE},\ \mathit{VE}_1 \uplus \mathit{VE}_2,\ \mathit{DE}_1,\ \mathit{CE}_1 \uplus \mathit{CE}_2)\\ \mathit{CON}(E) \subseteq \mathrm{Dom}\ \mathit{CE}_1\\ \mathit{VAR}(E) \subseteq \mathrm{Dom}\ \mathit{VE}_1\\ \Delta_2 = (\mathit{TE},\ \mathit{VE}_1,\ \mathit{DE}_1,\ \mathit{CE}_1)\end{array}}{\langle\!\langle \Delta_1,\ \overline{x}^k,\ E \rangle\!\rangle \Rightarrow_{gc} \langle\!\langle \Delta_2,\ \overline{x}^k,\ E \rangle\!\rangle} \tag{6.22}$$

Figure 6.4: *Closure Garbage Collection.*

The use of type information within the garbage collector enables a number of extensions to the collection algorithm. For brevity, the model presented in this chapter deals only with heap garbage collection. However, space leaks in the heap may also result from stack and environment garbage. Stack garbage results from recursive tail-calls, and environment garbage results from unused bindings. As an example, Figure 6.4 illustrates the removal of unused bindings from closure environments. *CON* returns the set of constructors which appear in E, and

VAR returns the set of variables which appear in E. This rule could readily be combined with Rule 6.18 to improve the efficiency of the garbage collection algorithm. The collection of stack garbage is a more complex problem, but could be tackled with a similar approach.

It is also worth noting that an alternative to garbage collection, called region based memory management, has recently been developed [TT97]. In this scheme, the memory is modelled using multiple stacks each containing values of a single type. A sophisticated region inference algorithm is used to determine the memory requirements of the program at compile-time, thereby avoiding the need for runtime garbage collection. Although this technique appears to be very promising, we will not make use of it here for a number of reasons. When using regions, the programmer must have a very good intuition about region inference to be able to construct programs without space leaks. This often requires the use of a complex profiling operation. Furthermore, the first-in last-out pattern of memory usage, arising from the use of multiple stacks, is often an inappropriate model of the life-time of dynamic data in a program. Thus, it may in fact be impossible to eliminate space-leaks in certain situations. It has been suggested that these problems can be overcome by combining region inference with garbage collection. However, this is outside the scope of this thesis. Our main reasons for rejecting the region based technique are more pragmatic. The definition of region inference contains a great number of rules, exceeding the definition of Standard ML in complexity. Thus, a great deal of effort would be needed to modify these rules for code replacement. In addition, region-based memory management is a very uncommon implementation technique in practice. Finally, the region inference technique is designed for sequential operation and contains no obvious method of integration into a distributed environment.

Chapter 7

Code Replacement

In this chapter we define a model of sequential code-replacement. We have laid the foundations for our code-replacement technique in the preceding chapters. However, before proceeding, we must first establish the association between code replacement and the Standard ML module system. The complete module system of [MTHM97] is somewhat daunting as it contains many complex though powerful features. To define code-replacement, it is necessary to make a number of changes to the published definition. The most important of these is the removal of transparent signature matching. The rationale behind this was discussed in detail in Chapter 1. To summarise, we cannot allow the internal types of the modules to be visible, since these types will change during replacement. While this restriction does reduce the power of the language, it encourages a good programming style making full use of abstraction. A similar restriction was made in the definition of the CAML Special Light dialect [Ler95].

Our treatment of the module system has been strongly influenced by [Els99]. In keeping with this thesis, we consider the module system purely as a compile-time framework. No runtime code for the module system is generated. This framework also permits the separate compilation of modules using standard compiler linking mechanisms. In the remainder of this chapter, we will briefly define the elaboration of the modules language in such a way as to ensure the correctness of code replacement. We will then show how the module-level constructs of the language may be removed. This operation results in a pure $\mathcal{M}\Lambda$ program which can be evaluated by the abstract machine of Chapter 5. Finally, we will define modular code-replacement as an extension of the garbage collection algorithm of Chapter 6. The central idea is to use the fact that the garbage collector already visits all of the live data of a program. With a number of relatively straightforward changes, the algorithm can perform an update of the program data to a new representation.

7.1 Dynamic ML Modules

The syntax for our Dynamic ML modules language is shown in Figure 7.1. This language introduces the identifiers *strid*, *sigid*, and *funid* for structures, signatures, and functors respectively. It also refers to a number of syntactic categories defined in Chapter 4. When compared with the modules grammar of Standard ML [MTHM97], the main omission is transparent signature matching, and the associated `sharing type` constraint. The only connection between the modules language and the core language is the declaration *dec* inside a structure *strdec*. We do not include other connections, such as `open` declarations, and `local` structures.

As we saw in Chapter 1, the all-or-nothing nature of opaque signature matching is undesirable in a number of code-replacement scenarios. Therefore, in addition to the Standard ML `where type` qualification, we include an additional `where datatype` qualification which allows the details of certain hidden datatype declarations to be constrained for replacement purposes.

program	::=	*topdec*	Modules Program
topdec	::=	*strdec*	Structure Declaration
	|	`functor` *funid* `(`*strid* `:` *sigexp*`)` `=` *strexp*	Functor Declaration
	|	$topdec_1$ ⟨`;`⟩ $topdec_2$	Sequence
strdec	::=	`structure` *strid* `=` *strexp*	Module Declaration
	|	*dec*	Core Declaration
	|	$strdec_1$ ⟨`;`⟩ $strdec_2$	Sequence
strexp	::=	`struct` *strdec* `end`	Structure Body
	|	*strid*	Structure Identifier
	|	*strexp* `:>` *sigexp*	Opaque Match
	|	*funid* `(`*strexp*`)`	Functor Application
sigexp	::=	`sig` *spec* `end`	Signature
	|	*sigexp* `where`	
		`type` `(`$tyvar_1$`,` $\cdots$ `,` $tyvar_n$`)` *tycon* `=` *ty*	Type Qualification
	|	*sigexp* `where` `datatype` *datbind*	Datatype Qualification
spec	::=	`val` *vid* `:` *ty*	
	|	`type` `(`$tyvar_1$`,` $\cdots$ `,` $tyvar_n$`)` *tycon*	
	|	`datatype` *datbind*	
	|	`exception` *vid* ⟨`of` *ty*⟩	
	|	`structure` *strid* `:` *sigexp*	
	|	$spec_1$ ⟨`;`⟩ $spec_2$	

Figure 7.1: *Dynamic ML Modules Grammar.*

7.1.1 Elaboration of Modules

We will now define the elaboration of the modules language as an extension of the elaboration of the core language presented in Chapter 4. The inference of types for functions and values in the core language is complemented by the inference of signatures for structures, but not functors, in the modules language. For brevity, we only present the type system for the modules language. We do not define the actual type checking algorithm in detail as we did for the core language.

Modules Environment	$\mathcal{E}$	::=	$(SE,\ \Gamma)$
Structures	SE	::=	$strid\ \mapsto\ \mathcal{E}$
Signature	Σ	::=	$(\overline{\overline{tn}})(\mathcal{E})$
Functor Signature	Φ	::=	$(\overline{\overline{tn}})(\mathcal{E},\ \Sigma)$
Functors	FE	::=	$funid\ \mapsto\ \Phi$
Basis	$\mathcal{B}$	::=	$(FE,\ \mathcal{E})$

Figure 7.2: *Modules Typing Environments.*

The typing environments (also called *semantic objects*) for the modules language are illustrated in Figure 7.2. ML modules, called structures, are identified by unique names *strid*. A structure may contain core-level declarations and/or other structures, but not functors. To represent the structures in a program, we introduce a modules environment $\mathcal{E}$ containing a core-environment Γ paired with a structure environment SE. The structure environment SE maps structure identifiers to module environments. Hence, a top-level module environment references a hierarchy of structures and their corresponding module-environments. It is worth noting that this is not a hierarchy in the object-oriented sense. Structures are just a packaging mechanism, and the structure hierarchy controls the package visibility. There is no notion of *inheritance* between structures.

Signatures assume the role of type-schemes in the modules system. However, while type schemes bind *type variables*, signatures bind *type names*. We write a signature Σ as $(\overline{\overline{tn}})(\mathcal{E})$ to indicate that the type names $\overline{\overline{tn}}$ are bound in the modules environment $\mathcal{E}$. When matching a signature against a structure, the bound type names are instantiated while the free type names must match exactly. Analogously to the notion of most general type, called the *principal type*, in the core language there is a notion of most general signature in the modules language. The *principal signature* for a structure is the most permissive one, allowing all of the definitions in the structure body to be seen outside. We define signature matching in more detail later in this chapter.

Functors are parameterised modules. The functor signature Φ is composed from an argument signature $(\overline{\overline{tn}})(\mathcal{E})$, and a result signature Σ. The functor environment FE maps functor identifiers $funid$ to functor signatures.

All of the phrases in the modules language are elaborated in a static basis $\mathcal{B}$. The basis performs the same role as the typing environment Γ in Chapters 3 and 4. For simplicity, we write $\Gamma \in \mathcal{B}$ to refer to the top-level typing environment Γ in $\mathcal{B}$. This avoids the need to decompose the basis. We also apply the usual set-theoretic operations to the basis as we did for the environment Γ in Chapter 4, e.g. $\mathcal{B} \cup \mathcal{E}$.

The typing judgements for the modules language are shown in Figure 7.3. We have adopted the style of [Els99], rather than [MTHM97] in that all of the modules judgements elaborate to signatures. This style makes the elaboration somewhat more consistent between the Dynamic ML core and modules languages; phrases in the core language elaborate to $\mathcal{M}\Lambda$ types and phrases in the modules language elaborate to $\mathcal{M}\Lambda$ signatures. However, it is worth noting that the meaning of the rules is preserved from [MTHM97] where appropriate. Before presenting the modules type system, we discuss what it means for a module to *match* a signature. This is only a brief outline. A detailed definition, together with a proof of principality, can be found in [MT91].

$\mathcal{B} \vdash program : (\overline{\overline{tn}})(\mathcal{B}')$	(program)
$\mathcal{B} \vdash topdec : (\overline{\overline{tn}})(\mathcal{B}')$	(top-level declaration)
$\mathcal{B} \vdash strdec : \Sigma$	(structure declaration)
$\mathcal{B} \vdash strexp : \Sigma$	(structure expression)
$\mathcal{B} \vdash sigexp : \Sigma$	(signature expression)
$\mathcal{B} \vdash spec : \Sigma$	(specification)

Figure 7.3: *Modules Typing Judgements.*

A structure (module) S matches a signature Σ if, for some S', S *enriches* S', and S' is an *instance* of Σ. We write this using the notation $\Sigma \geq S' \prec S$, where $\geq$ is an instantiation, and $\prec$ is an enrichment relation. This leaves us with the task of defining these relations. Note that we have already met instantiation and enrichment at the core language level. What we are defining here is an extension of these concepts to the modules language.

Instantiation is a mechanism for hiding the implementation details of the type components of a module. In order to be able to instantiate type names, we must first define the notion of a *realisation*. A type realisation is a finite map from type names to type schemes $\varphi = tn \overset{map}{\mapsto} \sigma$, such that tn and $\varphi(tn)$ have the same arity. The *support* Supp φ of a realisation is defined as the set of type names $tn \in \overline{\overline{tn}}$, for which $\varphi(tn) \neq tn$. The *yield* Yield φ of a realisation is the set of type names which occur in some $\varphi(tn)$ for which $tn \in$ Supp φ. For example, let tn_1 be a type name with arity 1, and tn_2 be a type name with arity 0. Then applying the realisation $\varphi = \{tn_1 \mapsto \forall\alpha.\ \alpha \rightarrow tn_2\}$ to the type $\tau = tn_1(tn_2)$ will result in the type $\varphi(\tau) = tn_2 \rightarrow tn_2$, where Supp $\varphi = \{tn_1\}$, and Yield $\varphi = \{tn_2\}$. Realisations are extended to apply to all semantic objects; their effect is to replace each type name tn by $\varphi(tn)$. In applying φ to to an object with bound names, such as a signature $(\overline{\overline{tn}})(\mathcal{E})$, the bound names must first be changed, so that for each binding prefix $(\overline{\overline{tn}})$, $\overline{\overline{tn}} \cap (\text{Supp } \varphi \cup \text{Yield } \varphi) = \emptyset$.

We can now define instantiation formally as follows: a modules environment $\mathcal{E}'$ is an instance of a signature $\Sigma = (\overline{\overline{tn}})(\mathcal{E})$, written $\Sigma \geq \mathcal{E}'$, if there exists a realisation φ such that $\varphi(\mathcal{E}) = \mathcal{E}'$, and Supp $\varphi \subseteq \overline{\overline{tn}}$. Instantiation also extends to functor signatures; given a functor signature $\Phi = (\overline{\overline{tn}})(\mathcal{E}_1,\ \Sigma_1)$, a functor instance $(\mathcal{E}_2,\ \Sigma_2)$ is an instance of Φ, written $\phi \geq (\mathcal{E}_2,\ \Sigma_2)$, if there exists a realisation φ such that $\varphi(\mathcal{E}_1,\ \Sigma_1) = (\mathcal{E}_2,\ \Sigma_2)$, and Supp $\varphi \subseteq \overline{\overline{tn}}$.

If S enriches S' then S is permitted to have more components and more polymorphism than S', though S and S' must agree on the names of structure and type components. Formally, an modules environment $\mathcal{E}_1 = (SE_1,\ \Gamma_1)$ enriches another modules environment $\mathcal{E}_2 = (SE_2,\ \Gamma_2)$, written $\mathcal{E}_1 \prec \mathcal{E}_2$, if all of the following conditions hold:

1. Dom $SE_1 \subseteq$ Dom SE_2 and $SE_1(strid) \prec SE_2(strid)$ for all $strid \in$ Dom SE_2.
2. Dom $(VE_1 \in \Gamma_1) \subseteq$ Dom $(VE_2 \in \Gamma_2)$ and $VE_1(x) \prec VE_2(x)$ for all $x \in$ Dom VE_2.
3. Dom $(CE_1 \in \Gamma_1) \subseteq$ Dom $(CE_2 \in \Gamma_2)$ and $CE_1(con) \prec CE_2(con)$ for all $con \in$ Dom CE_2.
4. Dom $(DE_1 \in \Gamma_1) \subseteq$ Dom $(DE_2 \in \Gamma_2)$ and $DE_1(tn) \prec DE_2(tn)$ for all $tn \in$ Dom DE_2, where $(n_1,\ \overline{\overline{con_1}}) \prec (n_2,\ \overline{\overline{con_2}})$ if:
   (a) $n_1 = n_2$
   (b) Either $\overline{\overline{con_1}} = \overline{\overline{con_2}}$, or $\overline{\overline{con_2}} = \emptyset$

Signature matching is therefore a combination of instantiation and enrichment. A module, represented by a module environment $\mathcal{E}$ matches a signature Σ iff there is another module environment $\mathcal{E}'$ such that $\Sigma \geq \mathcal{E}' \prec \mathcal{E}$. For the purpose of defining the type system, this definition is enough. However, what this definition does not define is how the environment $\mathcal{E}'$ is actually obtained. For this, we need to use a variant of algorithm $\mathcal{W}$, although we will not present the details.

Modules Program $\mathcal{B} \vdash program : (\overline{\overline{tn}})(\mathcal{B}')$

$$\frac{\mathcal{B} \vdash topdec : (\overline{\overline{tn}})(\mathcal{B}')}{\mathcal{B} \vdash topdec : (\overline{\overline{tn}})(\mathcal{B}')} \tag{7.1}$$

Top-level Declaration $\mathcal{B} \vdash topdec : (\overline{\overline{tn}})(\mathcal{B}')$

$$\frac{\mathcal{B} \vdash strdec : (\overline{\overline{tn}})(\mathcal{E})}{\mathcal{B} \vdash strdec : (\overline{\overline{tn}})(\emptyset,\ \mathcal{E})} \tag{7.2}$$

$$\frac{\begin{array}{l}\mathcal{B} \vdash sigexp : (\overline{\overline{tn}})(\mathcal{E}) \qquad \overline{\overline{tn}} \cap tnames(\mathcal{B}) = \emptyset \\ \mathcal{B}[strid\ \mapsto\ \mathcal{E}] \vdash strexp : \Sigma \qquad FE = \{funid\ \mapsto\ (\overline{\overline{tn}})(\mathcal{E},\ \Sigma)\}\end{array}}{\mathcal{B} \vdash \texttt{functor}\ funid\ \texttt{(}strid\ \texttt{:}\ sigexp\texttt{)}\ \texttt{=}\ strexp : (\emptyset)(FE,\ \emptyset)} \tag{7.3}$$

Comment: (Rule 7.3) The *tnames* function returns the set of all type names in $\mathcal{B}$. The condition $\overline{\overline{tn}} \cap tnames(\mathcal{B}) = \emptyset$ ensures that there is no accidental sharing between $\mathcal{E}$ and $\mathcal{B}$. The functor signature in FE is constructed using the elaborated *sigexp* as argument signature and the elaborated *strexp* as the result signature. The resulting set of bound type names is empty as it is functor application, not declaration, which generates new type names.

$$\frac{\begin{array}{l}\mathcal{B}_1 \vdash topdec_1 : (\overline{\overline{tn_1}})(\mathcal{B}_2) \qquad \mathcal{B}_1 \cup \mathcal{B}_2 \vdash topdec_2 : (\overline{\overline{tn_2}})(\mathcal{B}_3) \\ (\overline{\overline{tn_1}} \cup \overline{\overline{tn_2}}) \cap tnames(\mathcal{B}_1) = \emptyset \qquad \overline{\overline{tn_2}} \cap (\overline{\overline{tn_1}} \cup tnames(\mathcal{B}_2)) = \emptyset\end{array}}{\mathcal{B}_1 \vdash topdec_1\ \langle\texttt{;}\rangle\ topdec_2 : (\overline{\overline{tn_1}} \cup \overline{\overline{tn_2}})(\mathcal{B}_2 \cup \mathcal{B}_3)} \tag{7.4}$$

Comment: (Rule 7.4) The conditions on the type names in this rule ensure that type generativity is enforced i.e. no overlap between type names.

Structure Declaration $\mathcal{B} \vdash strdec : \Sigma$

$$\frac{\mathcal{B} \vdash strexp : (\overline{\overline{tn}})(\mathcal{E})}{\mathcal{B} \vdash \texttt{structure}\ strid\ \texttt{=}\ strexp : (\overline{\overline{tn}})(\{strid\ \mapsto\ \mathcal{E}\},\ \emptyset)} \tag{7.5}$$

$$\frac{\Gamma \in \mathcal{B} \vdash dec : \Gamma' \qquad \Gamma' = (TE,\ VE,\ DE,\ CE)}{\mathcal{B} \vdash dec : (\mathrm{Dom}\ DE)(\emptyset,\ \Gamma')} \tag{7.6}$$

Comment: (Rule 7.6) Elaborating a core declaration *dec* results in an environment Γ' containing all of the declarations in Γ together with the declarations of *dec*. The environment $\Gamma \in \mathcal{B}$ refers to the top-level environment in $\mathcal{B}$. The bound type variables are those which occur in the datatype environment DE of Γ'.

$$\frac{\begin{array}{ll} \mathcal{B} \vdash \mathit{strdec}_1 : (\overline{\overline{tn_1}})(\mathcal{E}_1) & \mathcal{B} \cup \mathcal{E}_1 \vdash \mathit{strdec}_2 : (\overline{\overline{tn_2}})(\mathcal{E}_2) \\ (\overline{\overline{tn_1}} \cup \overline{\overline{tn_2}}) \cap \mathit{tnames}(\mathcal{B}) = \emptyset & \overline{\overline{tn_2}} \cap (\overline{\overline{tn_1}} \cup \mathit{tnames}(\mathcal{E}_1)) = \emptyset \end{array}}{\mathcal{B} \vdash \mathit{strdec}_1 \ \langle \texttt{;} \rangle \ \mathit{strdec}_2 : (\overline{\overline{tn_1}} \cup \overline{\overline{tn_2}})(\mathcal{E}_1 \cup \mathcal{E}_2)} \tag{7.7}$$

Comment: (Rule 7.7) As in Rule 7.4, generativity is enforced by the conditions on the type names.

Structure Expression $\mathcal{B} \vdash \mathit{strexp} : \Sigma$

$$\frac{\mathcal{B} \vdash \mathit{strdec} : \Sigma}{\mathcal{B} \vdash \texttt{struct}\ \mathit{strdec}\ \texttt{end} : \Sigma} \tag{7.8}$$

$$\frac{\mathcal{B}(\mathit{strid}) = \mathcal{E}}{\mathcal{B} \vdash \mathit{strid} : (\emptyset)(\mathcal{E})} \tag{7.9}$$

Comment: (Rule 7.9) When referring to a structure by name, a signature is constructed from the modules environment of the structure in the basis. The set of bound type names is empty since we have not applied any signature constraints.

$$\frac{\begin{array}{ll} \mathcal{B} \vdash \mathit{strexp} : (\overline{\overline{tn_1}})(\mathcal{E}_1) & \mathcal{B} \vdash \mathit{sigexp} : (\overline{\overline{tn_2}})(\mathcal{E}_2) \\ (\overline{\overline{tn_2}})(\mathcal{E}_2) \geq \mathcal{E}_3 \prec \mathcal{E}_1 & \overline{\overline{tn_2}} \cap \mathit{tnames}(\mathcal{B}) = \emptyset \end{array}}{\mathcal{B} \vdash \mathit{strexp}\ \texttt{:>}\ \mathit{sigexp} : (\overline{\overline{tn_2}})(\mathcal{E}_2)} \tag{7.10}$$

Comment: (Rule 7.10) This rule checks that the structure, represented by the module environment $\mathcal{E}_1$, matches the signature $(\overline{\overline{tn_2}})(\mathcal{E}_2)$. This signature is also the overall result of the rule, due to opaque signature matching. If we were using transparent signature matching, then the result would be the signature $(\overline{\overline{tn_1}})(\mathcal{E}_1)$. The condition $\overline{\overline{tn_2}} \cap \mathit{tnames}(\mathcal{B}) = \emptyset$ ensures that there is no sharing between the signature and the basis.

$$\frac{\begin{array}{ll} \mathcal{B} \vdash \mathit{strexp} : (\overline{\overline{tn_1}})(\mathcal{E}_1) & \mathcal{B}(\mathit{funid}) \geq (\mathcal{E}_3,\ (\overline{\overline{tn_2}})(\mathcal{E}_2)) \\ \mathcal{E}_1 \succ \mathcal{E}_3 & (\overline{\overline{tn_1}} \cup \overline{\overline{tn_2}}) \cap \mathit{tnames}(\mathcal{B}) = \emptyset \end{array}}{\mathcal{B} \vdash \mathit{funid}\ \texttt{(}\mathit{strexp}\texttt{)} : (\overline{\overline{tn_1}} \cup \overline{\overline{tn_2}})(\mathcal{E}_2)} \tag{7.11}$$

Comment: (Rule 7.11) This rule is very similar to Rule 7.10; the structure *strexp* represented by the module environment $\mathcal{E}_1$ must match the argument part of the functor signature. However, the type names of the functor argument propagate to the result signature.

Signature Expression $\mathcal{B} \vdash \mathit{sigexp} : \Sigma$

$$\frac{\mathcal{B} \vdash \mathit{spec} : \Sigma}{\mathcal{B} \vdash \texttt{sig}\ \mathit{spec}\ \texttt{end} : \Sigma} \tag{7.12}$$

$$\frac{\begin{array}{l}\mathcal{B} \vdash \mathit{sigexp} : (\overline{\overline{tn_1}})(\mathcal{E}) \qquad \overline{\overline{tn_1}} \cap \mathit{tnames}(\mathcal{B}) = \emptyset \qquad \Gamma \in \mathcal{B} \vdash \mathit{ty} : \tau \\ \mathit{tycon} \leadsto tn_2 \qquad tn_2 \in \overline{\overline{tn_1}} \qquad \mathcal{E}(tn_2) = (0,\ \overline{\overline{con}}) \qquad \varphi = \{tn_2\ \mapsto\ \tau\}\end{array}}{\mathcal{B} \vdash \mathit{sigexp}\ \texttt{where type ()}\ \mathit{tycon}\ \texttt{=}\ \mathit{ty} : (\overline{\overline{tn_1}})(\varphi(\mathcal{E}))} \tag{7.13}$$

Comment: (Rule 7.13) A `where type` qualification is used to make an internal type of a module visible through an opaque signature, e.g. `structure intgraph :> GRAPH where type vertex = int`. This is achieved by applying a realisation φ, which exposes the type ty, over the opaque signature *sigexp*. To construct the realisation, the signature expression and types are converted into $\mathcal{M}\Lambda$ representations. We also check that the signature does not overlap with the basis, and that the signature type *tycon* is actually present in the signature and has arity 0.

$$\frac{\begin{array}{l}\mathcal{B} \vdash \mathit{sigexp} : (\overline{\overline{tn_1}})(\mathcal{E}) \qquad \overline{\overline{tn_1}} \cap \mathit{tnames}(\mathcal{B}) = \emptyset \qquad \Gamma \in \mathcal{B} \vdash \mathit{ty} : \tau \\ \mathit{tycon} \leadsto tn_2 \qquad tn_2 \in \overline{\overline{tn_1}} \qquad \mathcal{E}(tn_2) = (n,\ \overline{\overline{con}}) \\ \mathit{tyvar}_1 \leadsto \alpha^1\ \cdots\ \mathit{tyvar}_n \leadsto \alpha^n \qquad \varphi = \{tn_2\ \mapsto\ \forall\overline{\alpha}^n.\ \tau\}\end{array}}{\mathcal{B} \vdash \mathit{sigexp}\ \texttt{where type (}\mathit{tyvar}_1\texttt{,}\ \cdots\texttt{,}\ \mathit{tyvar}_n\texttt{)}\ \mathit{tycon}\ \texttt{=}\ \mathit{ty} : (\overline{\overline{tn_1}})(\varphi(\mathcal{E}))} \tag{7.14}$$

Comment: (Rule 7.14) The `where type` qualification can be used with polymorphism types. The type variables are supplied as arguments and checked against the arity of the signature type.

$$\frac{\begin{array}{l}\mathcal{B} \vdash \mathit{sigexp} : (\overline{\overline{tn_1}})(\mathcal{E}) \qquad \overline{\overline{tn_1}} \cap \mathit{tnames}(\mathcal{B}) = \emptyset \\ \Gamma \in \mathcal{B},\ (\overline{\overline{tn_1}})(\mathcal{E}) \vdash \mathit{datbind} : \Sigma\end{array}}{\mathcal{B} \vdash \mathit{sigexp}\ \texttt{where datatype}\ \mathit{datbind} : \Sigma} \tag{7.15}$$

Comment: (Rule 7.15) The `where datatype` qualification is required for code-replacement in Dynamic ML. In addition to exposing the internal types of a module, it also exposes the internal value constructors. A separate set of rules, which we define below, are required. These rules have the form $\Gamma,\ \Sigma_1 \vdash \mathit{datbind} : \Sigma_2$, where Σ_1 is the opaque signature, *datbind* is the definition of the datatypes which are to be made visible, and Σ_2 is the resulting signature.

Datatype Binding $\Gamma,\ \Sigma_1 \vdash \mathit{datbind} : \Sigma_2$

$$\frac{\mathit{tycon} \rightsquigarrow \mathit{tn}_2 \qquad \mathit{tn}_2 \in \overline{\overline{\mathit{tn}_1}} \qquad \mathcal{E}_1(\mathit{tn}_2) = (0,\ \emptyset) \qquad \Gamma,\ \mathit{tn}_2 \vdash \mathit{conbind} : \mathit{CE}}{\Gamma,\ (\overline{\overline{\mathit{tn}_1}})(\mathcal{E}_1) \vdash \texttt{()}\ \mathit{tycon}\ \texttt{=}\ \mathit{conbind} : (\overline{\overline{\mathit{tn}_1}})(\mathcal{E}_1[\mathit{tn}_2\ \mapsto\ (0,\ \mathrm{Dom}\ \mathit{CE})] \cup \mathit{CE})} \tag{7.16}$$

$$\frac{\begin{array}{l}\mathit{tycon} \rightsquigarrow \mathit{tn}_2 \qquad \mathit{tn}_2 \in \overline{\overline{\mathit{tn}_1}} \qquad \mathcal{E}_1(\mathit{tn}_2) = (0,\ \emptyset) \qquad \Gamma,\ \mathit{tn}_2 \vdash \mathit{conbind} : \mathit{CE} \\ \Gamma,\ (\overline{\overline{\mathit{tn}_1}})(\mathcal{E}_1[\mathit{tn}_2\ \mapsto\ (0,\ \mathrm{Dom}\ \mathit{CE})] \cup \mathit{CE}) \vdash \mathit{datbind} : (\overline{\overline{\mathit{tn}_1}})(\mathcal{E}_2)\end{array}}{\Gamma,\ (\overline{\overline{\mathit{tn}_1}})(\mathcal{E}_1) \vdash \texttt{()}\ \mathit{tycon}\ \texttt{=}\ \mathit{conbind}\ \texttt{and}\ \mathit{datbind} : (\overline{\overline{\mathit{tn}_1}})(\mathcal{E}_2)} \tag{7.17}$$

Comment: (Rules 7.16 and 7.16) These rules expose the constructors of a non-polymorphic datatype *datbind* in the signature $(\overline{\overline{\mathit{tn}_1}})(\mathcal{E}_1)$. The type constructor *tycon* is translated into a $\mathcal{M}\Lambda$ type name tn_2. Checks are then performed on the signature to ensure that the type name is bound, has arity 0, and initially has no constructors. The constructors *conbind* which we want to expose are translated into a constructor environment CE. This translation was defined in Chapter 4. The constructor environment is appended to the signature environment $\mathcal{E}_1$ to make the constructors visible. The entry for tn_2 in the datatype environment of $\mathcal{E}_1$ is also updated with the constructor names in CE. Finally, if there are any more datatypes (Rule 7.17) they are processed by a recursive invocation of the translation rule.

$$\frac{\begin{array}{l}\mathit{tycon} \rightsquigarrow \mathit{tn}_2 \qquad \mathit{tn}_2 \in \overline{\overline{\mathit{tn}_1}} \qquad \mathcal{E}(\mathit{tn}_2) = (n,\ \emptyset) \\ \mathit{tyvar}_1 \rightsquigarrow \alpha^1\ \cdots\ \mathit{tyvar}_n \rightsquigarrow \alpha^n \qquad \varphi = \{\mathit{tn}_2\ \mapsto\ \forall\overline{\alpha}^n.\ \mathit{tn}_2(\overline{\alpha}^n)\} \\ \Gamma,\ \mathit{tn}_2,\ \overline{\alpha}^n \vdash \mathit{conbind} : \mathit{CE}\end{array}}{\begin{array}{l}\Gamma,\ (\overline{\overline{\mathit{tn}_1}})(\mathcal{E}_1) \vdash \texttt{(}\mathit{tyvar}_1\texttt{,}\ \cdots\texttt{,}\ \mathit{tyvar}_n\texttt{)}\ \mathit{tycon}\ \texttt{=}\ \mathit{conbind} : \\ \quad (\overline{\overline{\mathit{tn}_1}})(\varphi(\mathcal{E}_1)[\mathit{tn}_2\ \mapsto\ (n,\ \mathrm{Dom}\ \mathit{CE})] \cup \mathit{CE})\end{array}} \tag{7.18}$$

$$\frac{\begin{array}{l}\mathit{tycon} \rightsquigarrow \mathit{tn}_2 \qquad \mathit{tn}_2 \in \overline{\overline{\mathit{tn}_1}} \qquad \mathcal{E}(\mathit{tn}_2) = (n,\ \emptyset) \\ \mathit{tyvar}_1 \rightsquigarrow \alpha^1\ \cdots\ \mathit{tyvar}_n \rightsquigarrow \alpha^n \qquad \varphi = \{\mathit{tn}_2\ \mapsto\ \forall\overline{\alpha}^n.\ \mathit{tn}_2(\overline{\alpha}^n)\} \\ \Gamma,\ \mathit{tn}_2,\ \overline{\alpha}^n \vdash \mathit{conbind} : \mathit{CE} \\ \Gamma,\ (\overline{\overline{\mathit{tn}_1}})(\varphi(\mathcal{E}_1)[\mathit{tn}_2\ \mapsto\ (n,\ \mathrm{Dom}\ \mathit{CE})] \cup \mathit{CE}) \vdash \mathit{datbind} : (\overline{\overline{\mathit{tn}_1}})(\mathcal{E}_2)\end{array}}{\Gamma,\ (\overline{\overline{\mathit{tn}_1}})(\mathcal{E}_1) \vdash \texttt{(}\mathit{tyvar}_1\texttt{,}\ \cdots\texttt{,}\ \mathit{tyvar}_n\texttt{)}\ \mathit{tycon}\ \texttt{=}\ \mathit{conbind}\ \texttt{and}\ \mathit{datbind} : (\overline{\overline{\mathit{tn}_1}})(\mathcal{E}_2)} \tag{7.19}$$

Comment: (Rules 7.18 and 7.19) Polymorphic datatypes are manipulated in a similar manner. However, after the translation of the type variables *tyvar* to $\mathcal{M}\Lambda$ type variables α, a check is performed to ensure that the arity of the type name n is equal to the number of type variables. A realisation φ is also constructed to expose the polymorphism of tn_2 in the signature.

Specification $\mathcal{B} \vdash spec : \Sigma$

$$\frac{vid \rightsquigarrow x \qquad \Gamma \in \mathcal{B} \vdash ty : \tau \qquad ftvs(\tau) = \emptyset}{\mathcal{B} \vdash \texttt{val}\ vid\ \texttt{:}\ ty : (\emptyset)(\{x \mapsto \tau\})} \tag{7.20}$$

$$\frac{vid \rightsquigarrow x \qquad \Gamma \in \mathcal{B} \vdash ty : \tau \qquad ftvs(\tau) \neq \emptyset}{\mathcal{B} \vdash \texttt{val}\ vid\ \texttt{:}\ ty : (\emptyset)(\{x \mapsto \forall ftvs(\tau).\ \tau\})} \tag{7.21}$$

Comment: (Rules 7.20 and 7.21) A value declaration in a signature specification is translated into an entry in the variable environment *VE* of the signature. A totally closed type scheme is created for polymorphic declarations (Rule 7.21).

$$\frac{tycon \rightsquigarrow tn}{\mathcal{B} \vdash \texttt{type ()}\ tycon : (\{tn\})(\{tn \mapsto (0,\ \emptyset)\})} \tag{7.22}$$

$$\frac{tyvar_1 \rightsquigarrow \alpha^1 \ \cdots\ tyvar_n \rightsquigarrow \alpha^n \qquad tycon \rightsquigarrow tn}{\mathcal{B} \vdash \texttt{type (}tyvar_1\texttt{,}\ \cdots\texttt{,}\ tyvar_n\texttt{)}\ tycon : (\{tn\})(\{tn \mapsto (n,\ \emptyset)\})} \tag{7.23}$$

Comment: (Rules 7.22 and 7.23) A signature type is translated into a bound type name in the signature. The arity of a polymorphic type is also recorded (Rule 7.23).

$$\frac{\Gamma \in \mathcal{B} \vdash datbind : (DE,\ CE)}{\mathcal{B} \vdash \texttt{datatype}\ datbind : (\mathrm{Dom}\ DE)(\emptyset,\ (\emptyset,\ \emptyset,\ DE,\ CE))} \tag{7.24}$$

Comment: (Rule 7.24) A datatype declaration is translated into a datatype environment and constructor environment by the translation rules of Chapter 4. All of the type names in *DE* are bound in the signature.

$$\frac{vid \rightsquigarrow con \qquad \langle \Gamma \in \mathcal{B} \vdash ty : \tau \qquad tvars(\tau) = \emptyset\rangle}{\mathcal{B} \vdash \texttt{exception}\ vid\ \langle\texttt{of}\ ty\rangle : (\emptyset),\ (\{con \mapsto \langle \tau \rightarrow\rangle\ \mathrm{t\_exn}\})} \tag{7.25}$$

Comment: (Rule 7.25) An exception is treated as a constructor of a special datatype t_exn.

$$\frac{\mathcal{B} \vdash sigexp : (\overline{\overline{tn}})(\mathcal{E})}{\mathcal{B} \vdash \texttt{structure}\ strid\ \texttt{:}\ sigexp : (\overline{\overline{tn}})(\{strid \mapsto \mathcal{E}\})} \tag{7.26}$$

$$\frac{\begin{array}{l}\mathcal{B} \vdash spec_1 : (\overline{\overline{tn_1}})(\mathcal{E}_1) \qquad \mathcal{B} \cup \mathcal{E}_1 \vdash spec_2 : (\overline{\overline{tn_2}})(\mathcal{E}_2) \qquad \mathrm{Dom}\ \mathcal{E}_1 \cap \mathrm{Dom}\ \mathcal{E}_1 = \emptyset \\ (\overline{\overline{tn_1}} \cup \overline{\overline{tn_2}}) \cap tnames(\mathcal{B}) = \emptyset \qquad \overline{\overline{tn_2}} \cap (\overline{\overline{tn_1}} \cup tnames(\mathcal{E}_1)) = \emptyset\end{array}}{\mathcal{B} \vdash spec_1\ \langle\texttt{;}\rangle\ spec_2 : (\overline{\overline{tn_1}} \cup \overline{\overline{tn_2}})(\mathcal{E}_1 \cup \mathcal{E}_2)} \tag{7.27}$$

7.2 Signature Replacement

There are two kinds of code replacement in Dynamic ML which are discussed in Chapter 1, namely *signature replacement*, and *module replacement*. Signature replacement allows more permissive signatures to replace more restrictive ones, e.g. extending a signature with an extra function specification. Module replacement allows the code of a module to be completely replaced by another. The only restriction being that the replacement module must match the same signature. Module replacement also requires all of the values which depend on the old module to be updated to the new representation. Consequently, module replacement is a significantly more complex operation than signature replacement.

In the following section we will define a translation from Dynamic ML to $\mathcal{M}\Lambda$ which removes all of the module-level details. This produces a program that is suitable for module replacement. However, this technique relies on the elaboration to ensure that modules correctly implement their signatures. Therefore, signature replacement must be performed at the elaboration stage. The name *signature replacement* is somewhat misleading as the replacement must not change any of the type information about visible types and values, except to make them more permissive i.e. polymorphic. Signature replacement is essentially a one-way operation and cannot be undone once applied. As noted in Chapter 1, the name *signature extension* is possibly more appropriate. In fact, the kinds of extensions that we wish to permit on signatures are precisely those provided by enrichment which we have already defined. Figure 7.4 defines the enrichment of signatures $\Sigma_1 \prec \Sigma_2$ in terms of environment enrichment.

$$\frac{\overline{\overline{tn_1}} \subseteq \overline{\overline{tn_2}} \qquad \mathcal{E}_1 \prec \mathcal{E}_2}{(\overline{\overline{tn_1}})(\mathcal{E}_1) \prec (\overline{\overline{tn_2}})(\mathcal{E}_2)} \tag{7.28}$$

Figure 7.4: *Signature Enrichment (replacement).*

Signature replacement will typically be performed after a module replacement: we replace a module with another one matching the same signature, and then at some later stage replace the signature with a more permissive one to expose additional functions which are present in the new module. When performing signature replacement it is necessary to recompile all affected modules against the new signature. In this thesis, we do not actually define the machinery by which signature replacement and module replacement are implemented, although we do discuss this issue briefly at the end of the chapter.

7.3 Translating Dynamic ML Modules to $\mathcal{M}\Lambda$

What follows is a sketch of the translation of a Dynamic ML program with module-level constructs into $\mathcal{M}\Lambda$. We assume that the program has been elaborated successfully. Hence, the type information present in the signatures is no longer required, just as type annotations in the core language were not used during the translation in Chapter 4. The translation presented here is only an outline. For simplicity we do not deal with a number of renaming issues as these would require a tedious, though straightforward modification of all of the translation rules in Chapter 4. We discuss these issues in more detail below. A comprehensive definition of the modules translation technique, called *static interpretation*, can be found in [Els99].

There are two basic steps performed in the translation of the module system: specialisation of functors, and flattening of structures. Functors are *specialised* into distinct structures at each application. Although this will yield a larger program, functors are not allowed to be recursive, and so the specialisation process will terminate. All of the resulting structures, together with those present in the program, are then flattened and collated into a single $\mathcal{M}\Lambda$ program. The result is a program with no module-level constructs. The modules language is essentially being treated as a compile-time linking language.

The translation described above produces an $\mathcal{M}\Lambda$ program suitable for evaluation by the abstract machine. However, an additional step must be performed for code-replacement. Recall from Chapter 1 that code-replacement is achieved by using special `Install` structures. These structures contain functions which are used to translate between the old and new data representations. Clearly, flattening these structures into the main program will not have the desired effect. Therefore, the translation step generates two programs: a $\mathcal{M}\Lambda$ program containing a flattened representation of the `Install` structures, and a main $\mathcal{M}\Lambda$ program containing a flattened representation of the remaining functors and structures. Evaluation will begin by applying the `Install` program to perform the code-replacement, followed by an evaluation of the main program.

The translation is defined by the rules given shown in Figure 7.5. The main $\mathcal{M}\Lambda$ program is denoted by P, and the program containing the `Install` structures is denoted by R. To express the functor specialisation operation, we use a functor translation environment $\mathcal{F} ::= \mathit{funid} \mapsto \mathit{strexp}$, which maps functor identifiers to functor bodies. The $\mathcal{M}\Lambda$ program is constructed cumulatively as the rules are applied; in each rule R_1 and P_1 are the $\mathcal{M}\Lambda$ programs before performing the translation, and R_2 and P_2 are the resulting $\mathcal{M}\Lambda$ programs after the translation.

$\mathcal{B},\ \mathcal{S} \vdash \mathit{program} \rightsquigarrow (P,\ R)$	(program)
$\mathcal{S},\ (\mathcal{F}_1,\ P_1,\ R_1) \vdash \mathit{topdec} \rightsquigarrow (\mathcal{F}_2,\ P_2,\ R_2)$	(top-level declarations)
$\mathcal{S},\ (\mathcal{F}_1,\ P_1,\ R_1) \vdash \mathit{strdec} \rightsquigarrow (\mathcal{F}_2,\ P_2,\ R_2)$	(structure declarations)
$\mathcal{S},\ (\mathcal{F}_1,\ P_1) \vdash \mathit{strexp} \rightsquigarrow (\mathcal{F}_2,\ P_2)$	(structure expressions)

Figure 7.5: *Module Translation Rules.*

Before defining the actual translation rules, we must deal with the issue of identifier renaming. In Chapters 2 and 4, we insisted on unique identifier names in $\mathcal{M}\Lambda$ and Dynamic ML programs. This allowed us to sidestep a number of complicated scoping issues. However, with the addition of the module system, we are forced to revisit the issue. The structure flattening technique described above will fold the identifiers of a program into a single name-space. Hence, identifiers belonging to different structures may potentially collide. It may appear that we can fix this problem by extending the restriction that all the identifiers of a program are unique, even those which reside in different structures. However, with a bit of thought it is clear that this will not work with functors. When functor specialisation is applied we will have multiple copies of the same structure and hence multiple declarations of the same identifier. A better solution is the use of *fully-qualified* identifiers. In Standard ML there is a class of *long* identifiers, defined syntactically as:

$$\begin{array}{rcll} \mathit{longx} & ::= & x & \\ & \mid & \mathit{strid}_1 . \cdots . \mathit{strid}_n . x & (n \geq 1) \end{array}$$

A long identifier may contain a list of structure identifiers which denote a hierarchy. This hierarchy can be used to distinguish between identifiers of the same name. A fully qualified identifier is a long identifier which contains the full hierarchy of structures from the top-level. Although we do not have long identifiers in $\mathcal{M}\Lambda$, we can avoid name space collisions by using a string representation of the equivalent fully quantified identifier. To illustrate this technique, we define a function which will return a set of all the visible variable identifiers in a program. This set contains string representations of the fully qualified variable names.

The function $\mathit{visibles}(\mathcal{B})$ defined below takes a basis as an argument and returns a pair of sets $(\overline{\overline{x}},\ \overline{\overline{x}}')$. The first set contains the variables which are defined outside `Install` structures, and the second set contains variables defined inside `Install` structures. The function is defined recursively, beginning with Rule 7.29. We assume that variables x are represented by strings, and we use dot-notation to indicate string concatenation. The *longid* parameter is used to accumulate the hierarchy of structure (and functor) identifiers.

$$\frac{\begin{array}{l} FE = \{funid_1 \ \mapsto\ \Phi_1,\ \ldots,\ funid_n \ \mapsto\ \Phi_n\} \\ \vdash visibles(funid_1,\ \Phi_1) = (\overline{\overline{x_n}},\ \overline{\overline{x_n}}') \quad \cdots \\ \quad \vdash visibles(funid_n,\ \Phi_n) = (\overline{\overline{x_n}},\ \overline{\overline{x_n}}') \\ \vdash visibles(\mathcal{E}) = (\overline{\overline{x_{n+1}}},\ \overline{\overline{x_{n+1}}}') \end{array}}{\vdash visibles(FE,\ \mathcal{E}) = (\overline{\overline{x_1}} \cup \cdots \cup \overline{\overline{x_{n+1}}},\ \overline{\overline{x_1}}' \cup \cdots \cup \overline{\overline{x_{n+1}}}')} \tag{7.29}$$

Comment: (Rule 7.29) All of the functor signatures Φ are examined for visible declarations, and the result is combined with the visible declarations in the modules environment $\mathcal{E}$.

$$\frac{\vdash visibles(longid,\ \Sigma) = (\overline{\overline{x}},\ \overline{\overline{x}}')}{\vdash visibles(longid,\ (\overline{\overline{tn}})(\mathcal{E},\ \Sigma)) = (\overline{\overline{x}},\ \overline{\overline{x}}')} \tag{7.30}$$

Comment: (Rule 7.30) The visible declarations of a functor signature are only those which are present in the result signature Σ.

$$\frac{\vdash visibles(longid,\ \mathcal{E}) = (\overline{\overline{x}},\ \overline{\overline{x}}')}{\vdash visibles(longid,\ (\overline{\overline{tn}})(\mathcal{E})) = (\overline{\overline{x}},\ \overline{\overline{x}}')} \tag{7.31}$$

$$\frac{\begin{array}{l} SE = \{strid_1 \ \mapsto\ \mathcal{E}_1,\ \ldots,\ strid_n \ \mapsto\ \mathcal{E}_n\} \\ \vdash visibles(longid,\ strid_1,\ \mathcal{E}_1) = (\overline{\overline{x_1}},\ \overline{\overline{x_1}}') \quad \cdots \\ \quad \vdash visibles(longid,\ strid_n,\ \mathcal{E}_n) = (\overline{\overline{x_n}},\ \overline{\overline{x_n}}') \\ \vdash visibles(longid,\ \Gamma) = (\overline{\overline{x_{n+1}}},\ \overline{\overline{x_{n+1}}}') \end{array}}{\vdash visibles(longid,\ (SE,\ \Gamma)) = (\overline{\overline{x_1}} \cup \cdots \cup \overline{\overline{x_{n+1}}},\ \overline{\overline{x_1}}' \cup \cdots \cup \overline{\overline{x_{n+1}}}')} \tag{7.32}$$

Comment: (Rule 7.32) A modules environment will contain visible declarations in both the structure environment *SE* and the core environment Γ.

$$\frac{strid = \texttt{Install} \quad \vdash visibles(longid.strid,\ \Gamma) = (\overline{\overline{x}})}{\vdash visibles(longid,\ strid,\ \mathcal{E}) = (\emptyset,\ \overline{\overline{x}})} \tag{7.33}$$

$$\frac{strid \neq \texttt{Install} \quad \vdash visibles(longid.strid,\ \Gamma) = (\overline{\overline{x}})}{\vdash visibles(longid,\ strid,\ \mathcal{E}) = (\overline{\overline{x}},\ \emptyset)} \tag{7.34}$$

Comment: (Rules 7.33 and 7.34) The visible declarations are added to one of the sets depending on whether the declaration is in an `Install` structure or not.

$$\frac{VE \in \Gamma = \{x_1 \ \mapsto\ \sigma_1,\ \ldots,\ x_k \ \mapsto\ \sigma_k\}}{\vdash visibles(longid,\ \Gamma) = \{longid.x_1,\ \ldots,\ longid.x_k\}} \tag{7.35}$$

Comment: (Rules 7.35) A set of variables is constructed from the variable environment *VE*.

Modules Program $\mathcal{B},\ \mathcal{S} \vdash program \rightsquigarrow (P,\ R)$

We present the modules translation technique below. Note that Rule 7.43 for translating core-level declarations is only an outline; it does not perform any renaming of the identifiers. Nonetheless, these rules provide a reasonable overview of modules translation.

$$\frac{\begin{array}{l}\vdash visibles(\mathcal{B}) = (\overline{\overline{x}},\ \overline{\overline{x}}') \qquad P_1 = (\emptyset,\ \emptyset,\ \textbf{tuple}\ (\textbf{var}\ x_1,\ \ldots,\ \textbf{var}\ \ x_k))\\ R_1 = (\emptyset,\ \emptyset,\ \textbf{tuple}\ (\textbf{var}\ x'_1,\ \ldots,\ \textbf{var}\ \ x'_k))\\ \mathcal{S},\ (\emptyset,\ P_1,\ R_1) \vdash topdec \rightsquigarrow (\mathcal{F},\ P_2,\ R_2)\end{array}}{\mathcal{B},\ \mathcal{S} \vdash topdec \rightsquigarrow (P_2,\ R_2)} \tag{7.36}$$

Comment: (Rule 7.36) The basis $\mathcal{B}$, and substitution $\mathcal{S}$ are the results of elaboration. Evaluating either P or R will result in a tuple containing all of the visible variable declarations.

$$\frac{\begin{array}{ll}\vdash visibles(\mathcal{B}) = (\overline{\overline{x}},\ \emptyset) & P_1 = (\emptyset,\ \emptyset,\ \textbf{tuple}\ (\textbf{var}\ x_1,\ \ldots,\ \textbf{var}\ \ x_k))\\ R_1 = (\emptyset,\ \emptyset,\ \textbf{scon}\ unit) & \mathcal{S},\ (\emptyset,\ P_1,\ R_1) \vdash topdec \rightsquigarrow (\mathcal{F},\ P_2,\ R_2)\end{array}}{\mathcal{B},\ \mathcal{S} \vdash topdec \rightsquigarrow P_2} \tag{7.37}$$

Comment: (Rule 7.37) We may have a program which does not contain any `Install` structures. In this case, the *visibles* function will return an empty set of replacement variables. We use a dummy $\mathcal{M}\Lambda$ program **scon** *unit* to avoid the need to define non-replacement variants of the remaining rules.

Top-level Declaration $\mathcal{S},\ (\mathcal{F}_1,\ P_1,\ R_1) \vdash topdec \rightsquigarrow (\mathcal{F}_2,\ P_2,\ R_2)$

$$\frac{\mathcal{S},\ (\mathcal{F}_1,\ P_1,\ R_1) \vdash strdec \rightsquigarrow (\mathcal{F}_2,\ P_2,\ R_2)}{\mathcal{S},\ (\mathcal{F}_1,\ P_1,\ R_1) \vdash strdec \rightsquigarrow (\mathcal{F}_2,\ P_2,\ R_2)} \tag{7.38}$$

$$\frac{}{\begin{array}{l}\mathcal{S},\ (\mathcal{F},\ P,\ R) \vdash \texttt{functor}\ funid\ \texttt{(}strid\ \texttt{:}\ sigexp\texttt{)}\ \texttt{=}\ strexp \rightsquigarrow\\ \quad (\mathcal{F} \cup \{funid\ \mapsto\ strexp\},\ P,\ R)\end{array}} \tag{7.39}$$

Comment: (Rule 7.39) Translating a functor declaration results in a new entry in the functor translation environment. No actual code for the functor is generated here.

$$\frac{\begin{array}{l}\mathcal{S},\ (\mathcal{F}_1,\ P_1,\ R_1) \vdash topdec_1 \rightsquigarrow (\mathcal{F}_2,\ P_2,\ R_2)\\ \mathcal{S},\ (\mathcal{F}_2,\ P_2,\ R_2) \vdash topdec_2 \rightsquigarrow (\mathcal{F}_3,\ P_3,\ R_3)\end{array}}{\mathcal{S},\ (\mathcal{F}_1,\ P_1,\ R_1) \vdash topdec_1\ \langle\texttt{;}\rangle\ topdec_2 \rightsquigarrow (\mathcal{F}_3,\ P_3,\ R_3)} \tag{7.40}$$

Structure Declaration $\mathcal{S},\ (\mathcal{F}_1,\ P_1,\ R_1) \vdash \mathit{strdec} \leadsto (\mathcal{F}_2,\ P_2,\ R_2)$

$$\frac{\mathit{strid} \neq \texttt{Install} \qquad \mathcal{S},\ (\mathcal{F}_1,\ P_1,\ R) \vdash \mathit{strexp} \leadsto (\mathcal{F}_2,\ P_2,\ R)}{\mathcal{S},\ (\mathcal{F}_1,\ P_1,\ R) \vdash \texttt{structure}\ \mathit{strid}\ \texttt{=}\ \mathit{strexp} \leadsto (\mathcal{F}_2,\ P_2,\ R)} \tag{7.41}$$

$$\frac{\mathit{strid} = \texttt{Install} \qquad \mathcal{S},\ (\mathcal{F}_1,\ R_1,\ P) \vdash \mathit{strexp} \leadsto (\mathcal{F}_2,\ R_2,\ P)}{\mathcal{S},\ (\mathcal{F}_1,\ P,\ R_1) \vdash \texttt{structure}\ \mathit{strid}\ \texttt{=}\ \mathit{strexp} \leadsto (\mathcal{F}_2,\ P,\ R_2)} \tag{7.42}$$

Comment: (Rule 7.41 and 7.42) A structure declaration is flattened. The structure identifier is used to determine whether P or R will contain the declaration.

$$\frac{P_1 = (\overline{\overline{D_1}},\ \overline{\overline{X_1}},\ E_1) \qquad \mathcal{S},\ E_1 \vdash \mathit{dec} \leadsto (\overline{\overline{D_2}},\ \overline{\overline{X_2}},\ E_2)}{\mathcal{S},\ (\mathcal{F},\ P_1,\ R) \vdash \mathit{dec} \leadsto (\mathcal{F},\ (\overline{\overline{D_1}} \cup \overline{\overline{D_2}},\ \overline{\overline{X_1}} \cup \overline{\overline{X_2}},\ E_2),\ R)} \tag{7.43}$$

Comment: (Rule 7.43) A core-level declaration is translated into a program by the rules in Chapter 4. The identifiers in *dec* should be renamed to fully-qualified form, though we do not give the details here.

$$\frac{\mathcal{S},\ (\mathcal{F}_1,\ P_1,\ R_1) \vdash \mathit{strdec}_1 \leadsto (\mathcal{F}_2,\ P_2,\ R_2) \qquad \mathcal{S},\ (\mathcal{F}_2,\ P_2,\ R_2) \vdash \mathit{strdec}_2 \leadsto (\mathcal{F}_3,\ P_3,\ R_3)}{\mathcal{S},\ (\mathcal{F}_1,\ P_1,\ R_1) \vdash \mathit{strdec}_1\ \langle\texttt{;}\rangle\ \mathit{strdec}_2 \leadsto (\mathcal{F}_3,\ P_3,\ R_3)} \tag{7.44}$$

Structure Expression $\mathcal{S},\ (\mathcal{F}_1,\ P_1,\ R_1) \vdash \mathit{strexp} \leadsto (\mathcal{F}_2,\ P_2,\ R_2)$

$$\frac{\mathcal{S},\ (\mathcal{F}_1,\ P_1,\ R_1) \vdash \mathit{strdec} \leadsto (\mathcal{F}_2,\ P_2,\ R_2)}{\mathcal{S},\ (\mathcal{F}_1,\ P_1) \vdash \texttt{struct}\ \mathit{strdec}\ \texttt{end} \leadsto (\mathcal{F}_2,\ P_2)} \tag{7.45}$$

$$\frac{}{\mathcal{S},\ (\mathcal{F},\ P,\ R) \vdash \mathit{strid} \leadsto (\mathcal{F},\ P,\ R)} \tag{7.46}$$

$$\frac{\mathcal{S},\ (\mathcal{F}_1,\ P_1,\ R_1) \vdash \mathit{strexp} \leadsto (\mathcal{F}_2,\ P_2,\ R_2)}{\mathcal{S},\ (\mathcal{F}_1,\ P_1,\ R_1) \vdash \mathit{strexp}\ \texttt{:>}\ \mathit{sigexp} \leadsto (\mathcal{F}_2,\ P_2,\ R_2)} \tag{7.47}$$

Comment: (Rule 7.47) We do not generate any actual code for signature matching. The validity of the match is ensured by the elaboration step.

$$\frac{\mathcal{S},\ (\mathcal{F}_1,\ P_1,\ R_1) \vdash \mathit{strexp}_1 \leadsto (\mathcal{F}_2,\ P_2,\ R_2) \qquad \mathcal{F}_2(\mathit{funid}) = \mathit{strexp}_2 \qquad \mathcal{S},\ (\mathcal{F}_2,\ P_2,\ R_2) \vdash \mathit{strexp}_2 \leadsto (\mathcal{F}_3,\ P_3,\ R_3)}{\mathcal{S},\ (\mathcal{F}_1,\ P_1,\ R_1) \vdash \mathit{funid}\ \texttt{(}\mathit{strexp}_1\texttt{)} \leadsto (\mathcal{F}_3,\ P_3,\ R_3)} \tag{7.48}$$

Comment: (Rule 7.48) A fresh copy of the functor is generated at each application. We assume a renaming in Rule 7.43 to prevent collisions between identifiers.

7.4 Runtime Module Replacement

In the remainder of this chapter, we extend the garbage collection algorithm from Chapter 6 with code-replacement. This is a refinement of the code-replacement operation which we have previously presented in [WKG98] and [WKG00]. The technique hinges on a semantic object which we call a replacement map, defined as follows:

$$\text{Replacement Map} \qquad RM \quad ::= \quad p_{old} \overset{map}{\mapsto} (l_{rep},\ p_{rep})$$

The domain of the replacement map is the set of the pointers to the types that are to be dynamically replaced. Each element p_{old} of the domain is mapped to a location and type-pointer pair $(l_{rep},\ p_{rep})$. The location contains the closure of the function which is to execute the replacement operation, and the type-pointer references the type which is to replace the old type.

Code-replacement is a straightforward extension of copying garbage collection. Any pointers or locations which are encountered during the collection are compared with those in the replacement map. If a match is found, then the replacement from the map is substituted during the copying operation. Since the garbage collection algorithm will visit all live data in a program, we can be assured that all old copies of the data will be replaced.

$$\frac{\begin{array}{l}
(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ R) \Rightarrow (H_2,\ \Delta_2,\ XS_2,\ (l_1,\ p_1) \cdot RS_1) \\
H_2(p_1) = \overline{p_2}^{\,k} \qquad H_2(l_1) = \overline{l_2}^{\,k} \qquad H_2(p_2^1) = p_3^1 \to p_4^1 \ \cdots \ H_2(p_2^k) = p_3^k \to p_4^k \\
RM = \{p_3^1 \mapsto (l_2^1,\ p_4^1),\ \ldots,\ p_3^k \mapsto (l_2^k,\ p_4^k)\} \\
(H_2,\ \emptyset,\ RM,\ \emptyset,\ \emptyset,\ \emptyset,\ \Delta_2) \Rightarrow_{\text{gc}} (Hf_1,\ Ht_1,\ RM,\ PF_1,\ LF_1,\ TE,\ \Delta_3) \\
(Hf_1,\ Ht_1,\ RM,\ PF_1,\ LF_1,\ TE,\ XS_2) \Rightarrow_{\text{gc}} \\
\quad (Hf_2,\ Ht_2,\ RM,\ PF_2,\ LF_2,\ TE,\ XS_3) \\
(Hf_2,\ Ht_2,\ RM,\ PF_2,\ LF_2,\ TE,\ RS_2) \Rightarrow_{\text{gc}} \\
\quad (Hf_3,\ Ht_3,\ RM,\ PF_3,\ LF_3,\ TE,\ RS_3) \\
(Ht_3,\ \Delta_3,\ XS_3,\ RS_3,\ P) \Rightarrow (H_4,\ \Delta_4,\ XS_4,\ (l_3,\ p_5) \cdot RS_4) \\
(Ht_4,\ \Delta_4,\ XS_4,\ RS_4,\ E) \Rightarrow (Ht_5,\ \Delta_5,\ XS_5,\ RS_5)
\end{array}}{(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ E) \Rightarrow_{\text{rep}} (H_5,\ \Delta_5,\ XS_5,\ RS_5)} \tag{7.49}$$

Figure 7.6: *Code Replacement with Garbage Collection.*

The code-replacement technique is defined in Figure 7.6. We assume that the abstract machine has been interrupted at a suitable point, i.e. while it is not evaluating any code which is to be replaced, though we do not give the details of how this is determined. The program P contains the code that will replace parts of the current program, and the program R contains the installation code which is used to convert from the old data representation to the new. We showed how P and R were obtained in the previous section. The replacement map is constructed by evaluating the replacement program R. A modified garbage collection is then performed from the roots to update the types and values to the new representations. The program P, containing the new code is then evaluated, and evaluation of the expression E is resumed.

There is a distinct advantage to using a copying garbage collection algorithm with code-replacement, as opposed to an in-place algorithm. In a copying collection, the *from* heap is preserved until the end, when it is discarded. Therefore, when performing code-replacement, we can abort the operation if an error occurs and roll-back to the previous state. An error is signalled by the raising of an uncaught exception within the code performing the replacement. The roll-back technique is illustrated in Figure 7.7. When an exception is raised, the GC abstract machine will **halt**. Note that the uncaught exception is referenced by $(l_3,\ p_5)$ and may be used to generate an error message. The *to* heap is then discarded to rollback the replacement, and evaluation resumes normally.

$$\frac{\begin{array}{l} (H_1,\ \Delta_1,\ XS_1,\ RS_1,\ R) \Rightarrow (H_2,\ \Delta_2,\ XS_2,\ (l_1,\ p_1)\cdot RS_1) \\ H_2(p_1) = \overline{p_2}^{\,k} \qquad H_2(l_1) = \overline{l_2}^{\,k} \qquad H_2(p_2^1) = p_3^1 \to p_4^1 \ \cdots \ H_2(p_2^k) = p_3^k \to p_4^k \\ RM = \{p_3^1 \ \mapsto\ (l_2^1,\ p_4^1),\ \ldots,\ p_3^k \ \mapsto\ (l_2^k,\ p_4^k)\} \\ (H_2,\ \emptyset,\ RM,\ \emptyset,\ \emptyset,\ \emptyset,\ \Delta_2) \Rightarrow_{\mathrm{gc}} \textbf{halt}\ (Hf,\ Ht,\ RM,\ PF_1,\ LF_1,\ TE_1,\ (l_3,\ p_5)) \\ (Hf,\ \Delta_1,\ XS_1,\ RS_1,\ E) \Rightarrow (H_3,\ \Delta_3,\ XS_3,\ RS_3) \end{array}}{(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ E) \Rightarrow_{\mathrm{rep}} (H_3,\ \Delta_3,\ XS_2,\ RS_3)} \tag{7.50}$$

Figure 7.7: *Rollback.*

The domain of the replacement map contains pointers to the types which are to be replaced. Ideally, the type heap would contain a unique allocation of each type, and therefore we could simply compare each pointer encountered during garbage collection against the replacement map. This was the assumption of the algorithm previously presented in [WKG00]. Unfortunately, the allocation functions τ_alloc, α_alloc, and σ_alloc of Chapter 5 performs a unique allocation of type names and type variables only. All other types, e.g. function types and polymorphic types, are freshly allocated on each invocation. This was done for simplicity since we have as yet been unable to construct an efficient algorithm for uniquely allocating types. Consequently, for replacement, we require an additional function to determine whether two different pointers refer to a representation of the same type. This is achieved by the predicate $H,\ M \vdash \mathit{pmatch}(p_1,\ p_2)$ defined below. The function checks for structural equivalence of types, corresponding to the equivalence of the static types defined in Chapter 3 (Figure 3.5). However, we allow any type variables to match any other. This allows us to compare type schemes without an expensive substitution operation. This is valid in the context of code-replacement as we will never be replacing a type variable which is not part of a type scheme. The set M stores pointers that have already been checked for equality, to prevent non-termination on recursive types. For convenience, we also define a wrapper function $H \vdash \mathit{rmatch}(p_1,\ RM) = p_2$ which searches for a match (using *pmatch*) for the type pointer p_1 in the replacement map RM, and returns the corresponding pointer p_2 from RM if a match is found.

$$\frac{H,\ \emptyset \vdash \mathit{pmatch}(p_1,\ p_2) \qquad p2 \in \mathrm{Dom}\ RM}{H \vdash \mathit{rmatch}(p_1,\ RM) = p_2} \tag{7.51}$$

$$\frac{p_1 \in M}{H,\ M \vdash \mathit{pmatch}(p_1,\ p_2)} \tag{7.52}$$

$$\frac{p_1 \notin M \qquad H(p_1) = \mathit{tn} \qquad H(p_2) = \mathit{tn}}{H,\ M \vdash \mathit{pmatch}(p_1,\ p_2)} \tag{7.53}$$

$$\frac{\begin{array}{l} p_1 \notin M \qquad H(p_1) = \mathit{tn}(p_3) \qquad H(p_2) = \mathit{tn}(p_4) \\ H,\ M \cup \{p_1\} \vdash \mathit{pmatch}(p_4,\ p_4) \end{array}}{H,\ M \vdash \mathit{pmatch}(p_1,\ p_2)} \tag{7.54}$$

$$\frac{\begin{array}{l} p_1 \notin M \qquad H(p_1) = \overline{p_3}^{\,k} \qquad H(p_2) = \overline{p_4}^{\,k} \\ H,\ M \cup \{p_1\} \vdash \mathit{pmatch}(p_3^1,\ p_4^1) \cdots H,\ M \cup \{p_1\} \vdash \mathit{pmatch}(p_3^k,\ p_4^k) \end{array}}{H,\ M \vdash \mathit{pmatch}(p_1,\ p_2)} \tag{7.55}$$

$$\frac{\begin{array}{l} p_1 \notin M \qquad H(p_1) = p_3 \to p_4 \qquad H(p_2) = p_5 \to p_6 \\ H,\ M \cup \{p_1\} \vdash \mathit{pmatch}(p_3,\ p_5) \qquad H,\ M \cup \{p_1\} \vdash \mathit{pmatch}(p_4,\ p_6) \end{array}}{H,\ M \vdash \mathit{pmatch}(p_1,\ p_2)} \tag{7.56}$$

$$\frac{p_1 \notin M \qquad H(p_1) = \alpha_1 \qquad H(p_2) = \alpha_2}{H,\ M \vdash \mathit{pmatch}(p_1,\ p_2)} \tag{7.57}$$

$$\frac{\begin{array}{l} p_1 \notin M \qquad H(p_1) = \langle\!\langle TE_1,\ p_3 \rangle\!\rangle \qquad H(p_2) = \langle\!\langle TE_2,\ p_4 \rangle\!\rangle \\ TE_1 = \{\alpha_1^1 \mapsto p_5^1,\ \ldots,\ \alpha_1^k \mapsto p_5^k\} \qquad TE_2 = \{\alpha_2^1 \mapsto p_6^1,\ \ldots,\ \alpha_2^k \mapsto p_6^k\} \\ H,\ M \cup \{p_1\} \vdash \mathit{pmatch}(p_5^1,\ p_6^1) \cdots H,\ M \cup \{p_1\} \vdash \mathit{pmatch}(p_5^k,\ p_6^k) \end{array}}{H,\ M \vdash \mathit{pmatch}(p_1,\ p_2)} \tag{7.58}$$

We will now define the code-replacement modifications to the garbage collection algorithm presented in Chapter 6. These alterations are remarkably straightforward, and essentially consist of just the three additional rules shown in Figure 7.8.

$$\frac{\begin{array}{l} p_1 \notin \mathrm{Dom}\ PF \qquad Hf_1 \vdash rmatch(p_1,\ RM) = p_2 \qquad RM(p_2) = (l_1,\ p_3) \\ (Hf_1,\ Ht_1,\ RM,\ PF_1,\ LF_1,\ TE_1,\ p_3) \Rightarrow_{\mathrm{gc}} \\ \quad (Hf_2,\ Ht_2,\ RM,\ PF_2,\ LF_2,\ TE_2,\ p_4) \end{array}}{\begin{array}{l} (Hf_1,\ Ht_1,\ RM,\ PF_1,\ LF_1,\ TE_1,\ p_1) \Rightarrow_{\mathrm{rep}} \\ \quad (Hf_2,\ Ht_2,\ RM,\ PF_2[p_1 \mapsto p_4],\ LF_2,\ TE_2,\ p_4) \end{array}} \tag{7.59}$$

$$\frac{\begin{array}{l} p_1 \notin \mathrm{Dom}\ PF \qquad Hf_1 \vdash rmatch(p_1,\ RM) = p_2 \\ RM(p_2) = (l_2,\ p_3) \qquad Hf_1(l_2) = \langle\!\langle \Delta_1,\ \{x\},\ E \rangle\!\rangle \\ (Hf_1,\ \Delta_1[x \mapsto (l_1,\ p_1)],\ (),\ (),\ E) \Rightarrow (Hf_2,\ \Delta_2,\ XS,\ (l_3,\ p_4) \cdot RS) \\ (Hf_2,\ Ht_1,\ PF_1,\ LF_1,\ TE_1,\ (l_3,\ p_4)) \Rightarrow_{\mathrm{gc}} \\ \quad (Hf_3,\ Ht_2,\ PF_2,\ LF_2,\ TE_2,\ (l_4,\ p_5)) \end{array}}{\begin{array}{l} (Hf_1,\ Ht_1,\ RM,\ PF_1,\ LF_1,\ TE_1,\ (l_1,\ p_1)) \Rightarrow_{\mathrm{rep}} \\ \quad (Hf_3,\ Ht_2,\ RM,\ PF_2,\ LF_2[l_1 \mapsto l_4],\ TE_2,\ (l_4,\ p_5)) \end{array}} \tag{7.60}$$

$$\frac{\begin{array}{l} p_1 \notin \mathrm{Dom}\ PF \qquad Hf_1 \vdash rmatch(p_1,\ RM) = p_2 \\ RM(p_2) = (l_2,\ p_3) \qquad Hf_1(l_2) = \langle\!\langle \Delta_1,\ \{x\},\ E \rangle\!\rangle \\ (Hf_1,\ \Delta_1[x \mapsto (l_1,\ p_1)],\ (),\ (),\ E) \Rightarrow \mathbf{halt}\ (Hf_2,\ \Delta_2,\ (),\ (l_3,\ p_4) \cdot RS) \end{array}}{\begin{array}{l} (Hf_1,\ Ht_1,\ RM,\ PF_1,\ LF_1,\ TE_1,\ (l_1,\ p_1)) \Rightarrow_{\mathrm{rep}} \\ \quad \mathbf{halt}\ (Hf_3,\ Ht_1,\ RM,\ PF_1,\ LF_1,\ TE_1,\ (l_3,\ p_4)) \end{array}} \tag{7.61}$$

Figure 7.8: *Replacement of Types and Values.*

Rule 7.59 defines the replacement of type information. The pointer p_1 is compared with the replacement map using the *rmatch* function which we previously defined. If a match is not found, then the regular garbage collection rules will be applied, otherwise the remainder of the rule will be evaluated. The replacement type p_3 is obtained from the replacement map RM. The closure referenced by l_1 is not required at this stage. Note that the replacement type is still in the *from* heap Hf. The garbage collector is then invoked on the replacement type to copy it into the *to* heap, and the resulting pointer p_4 is returned as the final result. The forwarding table PF is updated to reflect the replacement of p_1 with p_4. The effect of the rule is the substitution of the old type with the new type. The old type remains in the *from* heap and will be removed upon completion of collection.

Replacing values requires more effort, as a function must be evaluated to convert between old and new data representations. Rules 7.60 and 7.61 define the replacement of a value referenced by $(l_1,\ p_1)$. In both rules, a closure $\langle\!\langle \Delta_1,\ \{x\},\ E \rangle\!\rangle$ is obtained from the replacement map to perform the conversion. This closure is evaluated by the $\mathcal{M}\Lambda$ abstract machine defined in Chapter 5, by binding $(l_1,\ p_1)$ to the argument variable x in the closure environment Δ_1. If the evaluation is successful, as defined in Rule 7.60, then the replacement value $(l_3,\ p_4)$ is returned, and garbage collected as before. However, if an uncaught exception is raised during the evaluation of Rule 7.61, the replacement will be halted immediately, and a rollback will occur in Rule 7.50.

There are two minor changes to the GC abstract machine that must be made in order to support code-replacement. We will not define these changes in detail as they are largely trivial. However, for completeness we will discuss them here. The state of the GC abstract machine must be modified in each of the garbage collection rules to include the replacement map. This is a very straightforward change as the replacement map is not modified in any of the rules and can therefore be simply passed around. The other change involves the *from* heap. Previously, this heap remained completely unchanged by garbage collection. However, with code replacement, the *from* heap may be extended when evaluating the closure in Rules 7.60 and 7.61. The garbage collection rules must therefore be updated to reflect the fact that the *from* heap may change when collecting values. This only requires a minor change of numbering in the rules. Note that no part of the *from* heap will be overwritten by the evaluation of the closure, and thus the rollback operation is valid. Any values which are placed into the *from* heap before rollback are simply treated as garbage and will be removed at the next collection.

The replacement model that we have now defined is very powerful. Any value whose type can be uniquely matched by a type in the domain of the replacement map *RM* can be replaced, and anything that can be expressed as a Dynamic ML function can be used in this replacement. In particular, data structures can be completely reworked, with both the addition and removal of constructors permitted. However, it is important to note that only type safety is guaranteed. The program is guaranteed to execute after replacement, but it is perfectly possible that the resulting program could be incorrect, i.e. there is no attempt made to preserve the semantic correctness of the program. Similarly, if a replacement is made to correct a flaw in a program, then the programmer must be aware that data generated by this flawed code may still be in the program, e.g. it has been used in a calculation, and must take this into account.

7.5 An Example Replacement

We will now present an example which illustrate our modules replacement technique in practice using the definitions which we have presented in this chapter. This example is based on the table replacement example which we previously presented in Chapter 1. In this introductory chapter we showed at a purely syntactic level how to replace the implementation of a abstract datatype for representing tables with another more efficient implementation. Initially, the table was represented as a simple unsorted list. In the replacement, the table was represented as an ordered binary tree.

As we have previously discussed in this chapter, there are a number of stages involved in performing code-replacement. We begin by constructing a module which contains the new implementation, together with a number of functions for converting values of the old representation into values of the new representation. The module must then pass through three phases, each of which will either result in the module being rejected, or the module continuing to the next phase. The successful completion of the final phase corresponds to the completion of the code-replacement.

The first phase of code-replacement is the elaboration phase. At this stage, a type checking operation is performed on the module to guarantee type safety. By ensuring that the module matches the same signature as the module which it replaces, we show that the module is indeed a valid candidate for replacement. Once the module has been successfully elaborated, a translation is performed to remove all of the module-level constructs. We have argued that these are merely packaging constraints and are not required for evaluation. At the same time we also construct a replacement map which we will use in the replacement of values. This code will be discarded upon conclusion of the replacement. Finally, we perform the actual code replacement. This is achieved by initiating a copying garbage collection. The actual replacement is performed on-the-fly during the copying operation. The result is that all the values and types of the old representation are replaced with the new representation. The old functions are then overwritten with their replacements and evaluation resumes.

We will now present a sequence of three small examples to illustrate each of these three phases in turn. We do not present the entire replacement in detail as it involves the complex interaction between elaboration, translation, evaluation, and garbage collection. However, the details shown should give the reader a good flavour of the complex processes involved.

Our first example illustrates the elaboration of a signature, as required for valid code replacement. In our example, any module which is to be considered a valid replacement for the table abstract data type must match the `TABLE` signature. This signature is shown as Dynamic ML code in Figure 7.9.

```
signature TABLE =
sig
  type () table
  val empty : () table
  val insert : {1 = string, 2 = () table} -> () table
  val member : {1 = string, 2 = () table} -> bool
end
```

Figure 7.9: *The* `TABLE` *signature in Dynamic ML.*

The elaboration process for the `TABLE` signature is shown in detail in Figure 7.11. The rules invoked at each stage are shown in parenthesis. Note that some rules from Chapter 4 are used. The result is the signature Σ shown in Figure 7.10. As we would expect, this signature contains a single bound type *table*, an *empty* value of type *table*, and a pair of functions *insert* and *member*.

$$\begin{array}{lll} \Sigma & = & (\{\mathit{table}\})\ \ (\{\mathit{table} \mapsto (0,\, \emptyset),\ \mathit{empty} \mapsto \mathit{table}, \\ & & \qquad \mathit{insert} \mapsto (\text{t\_string},\ \mathit{table}) \rightarrow \mathit{table}, \\ & & \qquad \mathit{member} \mapsto (\text{t\_string},\ \mathit{table}) \rightarrow \text{t\_bool}\}) \end{array}$$

Figure 7.10: *Elaborated signature* Σ*.*

Our second example illustrates the translation of a module-level program into one containing only core-level declarations. We translate the `InstallTable` functor from Chapter 1 into core-level programs P and R, where P contains the new table code, and R contains code to perform the update of the old table values. The `InstallTable` functor is shown as Dynamic ML code in Figure 7.12. In principle, the effect of evaluating this code will be the replacement of the `OldTable` module with `NewTable`, under the condition that both match the `TABLE` signature given above. For brevity we have replaced the core-level declarations in the functor with comments as we are only interested in the modules system here. The core-level translation was the subject of Chapter 4. Though we note that in an implementation, the translation mechanisms at these two-levels would be combined.

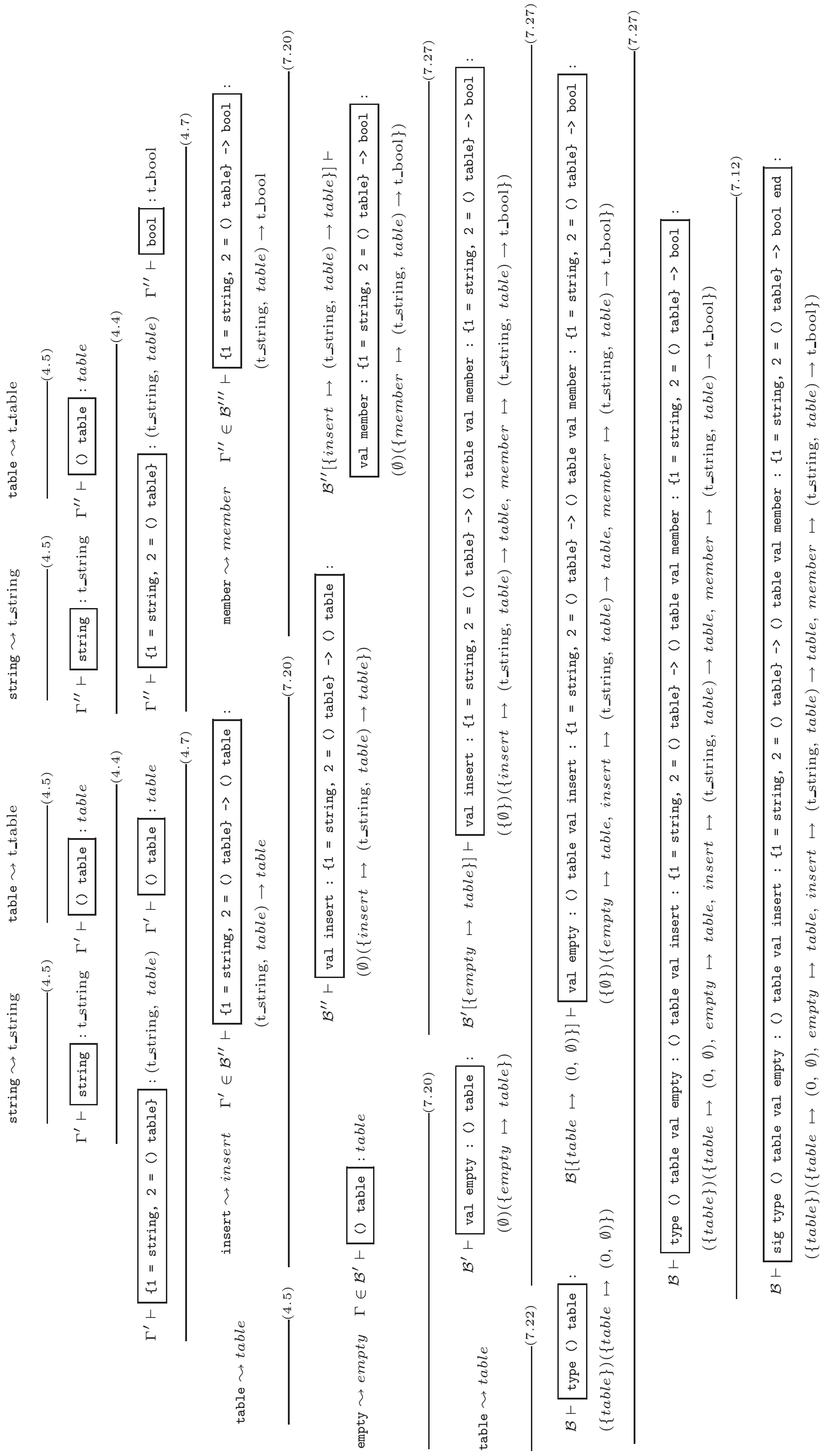


Figure 7.11: *Signature Inference.*

```
functor InstallTable(structure Table : TABLE where
                     type () table = (string) list) =
struct
  (* decs1 *)
  structure Install =
  struct
    (* decs2 *)
  end
end
structure NewTable = InstallTable(OldTable) :> TABLE;
```

Figure 7.12: *Replacement Module.*

The translation (specialisation) of the table functor into core-level declarations is shown in Figure 7.15. We assume that the modules program has already elaborated correctly, we also assume the existence of a translation rule of the form $P \vdash$ `(* decs *)` $\rightsquigarrow$ **let** *decs* **in** P for translating core-level declarations. The resulting programs P and R are shown as the tuple in Figure 7.13. Again, as we would expect, all module-level declarations have been removed in the translation and we are left with four functions. The functions *insert* and *member* in P are the new functions for table insertion and membership. The functions *name* and *table* in R will be used to translate the module types between representations.

$$\begin{array}{rcl}(P,\ R) & = & (\textbf{let}\ \mathit{decs}1\ \textbf{in tuple}\ (\textbf{var}\ \mathit{insert},\ \textbf{var}\ \mathit{member}), \\ & & \ \textbf{let}\ \mathit{decs}2\ \textbf{in tuple}\ (\textbf{var}\ \mathit{name},\ \textbf{var}\ \mathit{table}))\end{array}$$

Figure 7.13: *Translated programs P and R.*

In our final example, we illustrate the effect of code replacement on the program heap. At this stage, we have elaborated and specialised the modules in our example and we are only concerned with the actual data representations. The representations involved in the replacement are shown as $\mathcal{M}\Lambda$ datatypes in Figure 7.14. Clearly, the first representation is a list of names, while the second is a binary tree.

datatype (t_table) **of** {c_nil, (c_cons, (t_name, t_table))}
datatype (t_table) **of** {c_empty, (c_node, (t_table, t_name, t_table))}

Figure 7.14: *Table Data Representations.*

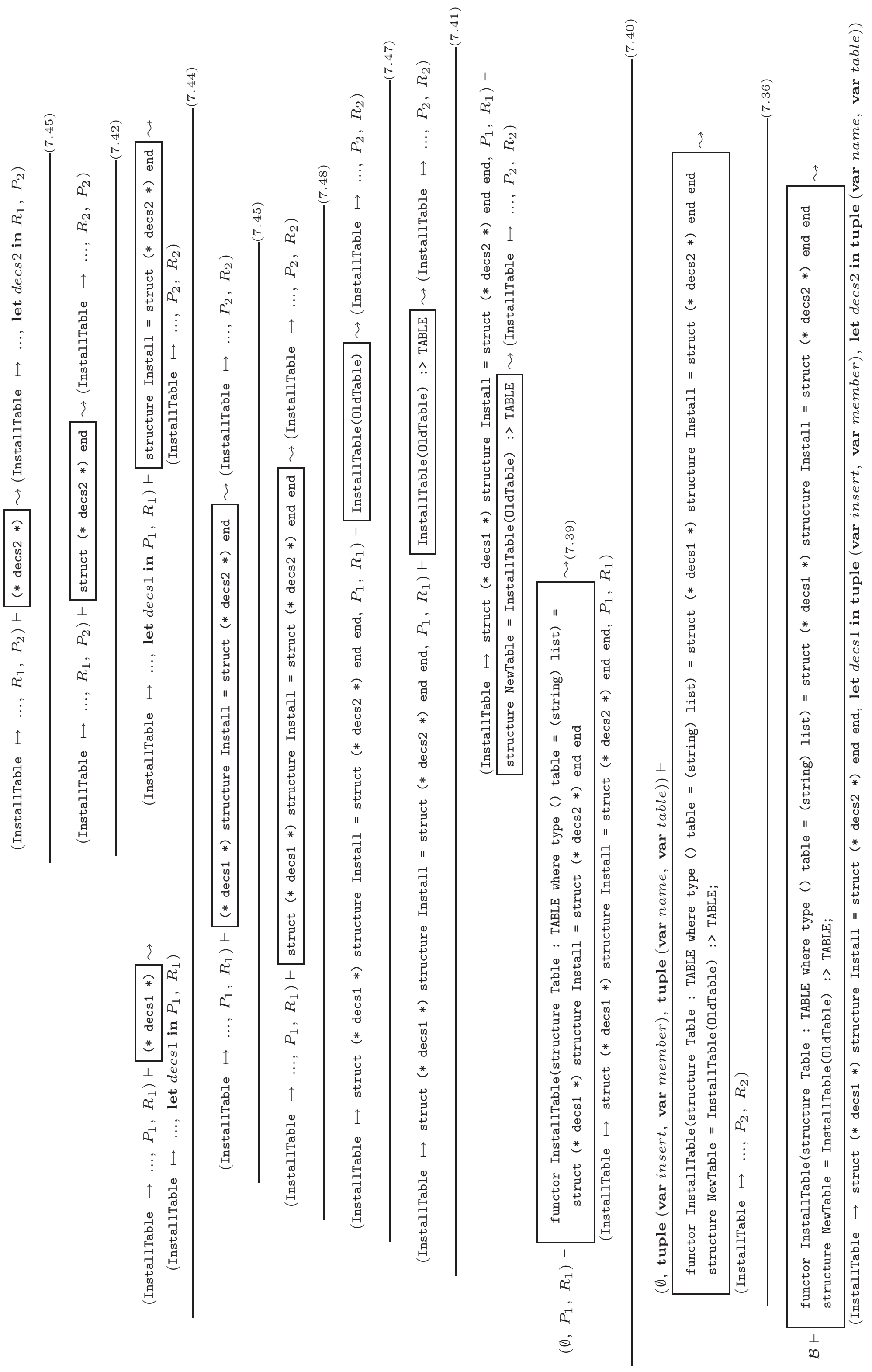


Figure 7.15: *Functor Specialisation.*

We will illustrate the effect of code replacement on an example heap shown in Figure 7.16. For now, we are only interested in the left side of this diagram, which shows the contents of the type heap and value heap prior to code replacement. At this point, the value heap contains a list of names of colours and the type heap contains types for all the different kinds of values. The colour names (l_0 to l_6) correspond to the type p_0, the c_nil constructor corresponds to type p_1, the pairs (l_9, l_{11}, ...) correspond to the type p_3, and the c_cons constructor corresponds to the type p_4.

Before Replacement				After Replacement			
Type Heap TH		Value Heap VH		Type Heap TH		Value Heap VH	
p_0	t_name	l_0	`"red"`	p_0	t_name	l_0	`"red"`
p_1	t_table	l_1	`"orange"`	p_1	t_table	l_1	`"orange"`
p_2	$(p_0,\ p_1)$	l_2	`"yellow"`	p_2	$(p_1,\ p_0,\ p_1)$	l_2	`"yellow"`
p_3	t_table(p_2)	l_3	`"green"`	p_3	t_table(p_2)	l_3	`"green"`
p_4	$p_3 \rightarrow p_1$	l_4	`"blue"`	p_4	$p_3 \rightarrow p_1$	l_4	`"blue"`
$\mathbf{p_5}$	**t_name**	l_5	`"indigo"`			l_5	`"indigo"`
$\mathbf{p_6}$	**t_table**	l_6	`"violet"`			l_6	`"violet"`
$\mathbf{p_7}$	$(\mathbf{p_6},\ \mathbf{p_5},\ \mathbf{p_6})$	l_7	c_nil			l_7	c_empty
$\mathbf{p_8}$	**t_table**($\mathbf{p_7}$)	l_8	c_cons(l_9)			l_8	c_node(l_9)
$\mathbf{p_9}$	$\mathbf{p_8} \rightarrow \mathbf{p_6}$	l_9	$(l_0,\ l_7)$			l_9	$(l_{10},\ l_0,\ l_{12})$
		l_{10}	c_cons(l_{11})			l_{10}	c_node(l_{11})
		l_{11}	$(l_1,\ l_8)$			l_{11}	$(l_{14},\ l_1,\ l_{18})$
		l_{12}	c_cons(l_{13})			l_{12}	c_node(l_{13})
		l_{13}	$(l_2,\ l_{10})$			l_{13}	$(l_{20},\ l_2,\ l_7)$
		l_{14}	c_cons(l_{15})			l_{14}	c_node(l_{15})
		l_{15}	$(l_3,\ l_{12})$			l_{15}	$(l_{16},\ l_3,\ l_7)$
		l_{16}	c_cons(l_{17})			l_{16}	c_node(l_{17})
		l_{17}	$(l_4,\ l_{14})$			l_{17}	$(l_7,\ l_4,\ l_7)$
		l_{18}	c_cons(l_{19})			l_{18}	c_node(l_{19})
		l_{19}	$(l_5,\ l_{16})$			l_{19}	$(l_7,\ l_5,\ l_7)$
		l_{20}	c_cons(l_{21})			l_{20}	c_node(l_{21})
		l_{21}	$(l_6,\ l_{18})$			l_{21}	$(l_7,\ l_6,\ l_7)$
		$\mathbf{l_{22}}$	⟪`fn x => x`⟫				
		$\mathbf{l_{23}}$	⟪`List.foldr insert empty`⟫				

Figure 7.16: *Heap Transformation.*

The first stage in the replacement of data values is the evaluation of the replacement program R and the construction of the replacement map RM. This evaluation occurs inside the original heap, before the garbage collection is initiated. The effect of this evaluation is shown in our example heap by the types p_5 through p_9 representing the tree data structure, and the values l_{22} and l_{23} which contain the closures for code replacement.

The replacement map for our example heap is built as follows:

$$RM = \{p_0 \rightarrow (l_{22},\ p_5),\ p_1 \rightarrow (l_{23},\ p_6)\}$$

The first component of this map will be used for converting the type t_name, while the second component will be used for converting the type t_table. The closures l_{22} and l_{23} correspond to the *name* and *table* functions in Figure 7.13 respectively.

Code replacement proceeds by performing a garbage collection on the heap and replacing all of the values whose types match those in the replacement map. The layout of our example heap after this operation is illustrated on the right side of Figure 7.16. A pictorial representation of the replacement process is also shown in Figure 7.17.

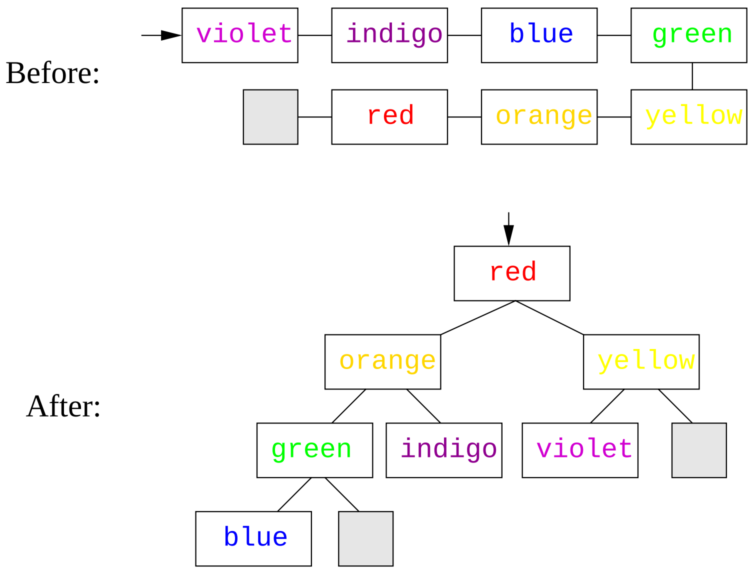


Figure 7.17: *Conversion of Data Representations.*

The important points to note about the heap after replacement are that all the old types, old data representations, and the functions used in the replacement operation are absent from the new heap. Therefore, there is no danger of any interference from the previous representations upon completion of replacement. It is also worth observing that the contents of the old heap is preserved until replacement completes. This allows for a roll-back operation, should any problems arise during replacement. Although the new types and replacement functions are placed in the old heap, they would be eliminated by a regular garbage collection operation should a roll-back occur.

7.6 Further Work

In this chapter we have presented a detailed definition of code-replacement in Dynamic ML. However, there are two areas of replacement that we have left undefined. The reason for these exclusions is primarily because they are implementation dependent. We will now describe these areas and outline potential solutions.

Closure Replacement

Our code-replacement technique defined in this chapter contains only a partial definition of the replacement of function closures. At present, the closure generated by the new function declaration simply overwrites the old function closure in the heap. However, this is inappropriate when the closure in the heap is the result of a partial function application. The requirement for correct replacement in this instance is for the closure to be updated with the new function code and variable bindings, i.e. a replacement of the following form:

$$\langle\!\langle \Delta_{old},\ \overline{x_{old}}^{k},\ E_{old} \rangle\!\rangle \Rightarrow_{rep} \langle\!\langle \Delta_{new},\ \overline{x_{new}}^{k},\ E_{new} \rangle\!\rangle$$

Unfortunately, this is a particularly difficult problem as the replacement cannot be uniquely determined using type information alone. In particular, the closure does not contain any information about the function from which it originated. We cannot simply pattern match between function expressions, since there may be many functions which match a particular expression, e.g. a partially applied `map`. Furthermore, the closure expression E_{old} is typically a piece of compiled machine code in an actual implementation. Therefore, we presently make the restriction that partially applicable functions cannot be replaced with our algorithm. Though, this is more restrictive than we would like.

In order to replace partially-applied functions, we need to resort to a tag (or pointer) in each closure to its parent function. We can then define a replacement map between function names and expressions. We note that the tagging operation is very similar to that required for symbolic debugging of functional programs, where one typically needs to examine the contents of closures at runtime. A scheme for tagging and opening closures in this manner is described in [App88]. This scheme is also applicable when the closure has been compiled into machine-code. However, we have not provided a definition here as this scheme is largely implementation-dependent and would considerably complicate our model.

Meta-Level Replacement

The other omission from our model concerns the initiation of replacement. We have left undefined the actual process by which modules are compiled and introduced into a running system. For example, in Figure 7.6, the programs P and R are used without stating how they came to be available. The underlying assumption in this chapter is that there is a meta-level which performs the following tasks:

1. Separate compilation of modules.
2. Interruption of the evaluation at appropriate points for replacement.
3. Initiation of garbage collection and replacement when necessary.

Separate compilation is the process of type checking and compiling individual modules in isolation. A good summary of modularisation and the issues involved in separate compilation can be found in [Ler94]. There have been many different schemes for performing separate compilation in Standard ML. Two of the most recent and relevant schemes can be found in [Rus98], and [Els99]. However, we note that most of the complexity in these schemes is related to transparent signature matching which we have removed from Dynamic ML. In Dynamic ML a scheme similar to a typical linker in an imperative language can be used, since all of the necessary information is contained directly in the signatures. We do not go into further detail on the composition of the meta-level as the issues involved are again largely implementation dependent and therefore tangential to our main task of providing a formal definition of code replacement.

Chapter 8

Distributed Evaluation

The definition of code-replacement in the preceding chapter assumes a sequential model of evaluation. In this chapter we lay the foundations for a distributed implementation of code-replacement. In this setting, the utility of the code-replacement technique becomes significantly more apparent. In the sequential setting, code replacement is primarily useful as a debugging tool. However, in the distributed setting, where large programs can be constructed spanning many machines, it becomes significantly more important. The provision of distributed code-replacement is considerably more challenging than in the sequential case. We must first develop a substantially expanded model for the dynamic semantics of the language which takes into account communication and concurrency, before we can begin to define code-replacement. A suitable abstract-machine model of the distributed language is the subject of this chapter.

$$\begin{array}{rcll}
E \in \text{Expression} & ::= & \mathbf{fork}\ E & \text{(local thread)} \\
 & | & \mathbf{rfork}\ (E_1,\ E_2) & \text{(remote thread)} \\
 & | & \mathbf{send}\ (E_1,\ E_2) & \text{(send value)} \\
 & | & \mathbf{receive}\ E & \text{(receive value)}
\end{array}$$

Figure 8.1: $\mathcal{M}\Lambda$ *Thread Expressions.*

In Chapter 2 we met all of the primitives in the $\mathcal{M}\Lambda$ language which are necessary for distributed evaluation. For convenience, these operations are summarised in Figure 8.1. The **fork** and **rfork** expressions give us the ability to launch threads, which are functions of type t_unit → t_unit, on local and remote processors respectively. The **send** and **receive** expressions give us the ability to communicate between these threads. These deceptively simple primitives enable the creation of powerful distributed systems on multiple processors with complex communication behaviours.

8.1 The LEMMA Interface

Before proceeding with a definition of distributed evaluation it is necessary to consider the structure and representation of the *memory* of the distributed system. We will make use the LEMMA (Laas and Edinburgh ML Memory Architecture) interface [MS95b], a general-purpose memory platform for distributed programming languages. The LEMMA interface has been designed to be independent of the details of the language implemented on it, and also as independent as possible of the system on which it is implemented. Nonetheless, the LEMMA interface contains a number of extensions which are specifically designed for the efficient support of the ML family of languages.

The LEMMA interface identifies two main memory services that are required in distributed systems: sharing of distributed data, and distributed garbage collection. The first of these is of primary importance in this chapter since it determines the structure of the memory. We define the garbage collection aspects of the LEMMA interface in Chapter 9 along with code-replacement. The LEMMA interface definition is simply a function-level specification, the actual implementation is left open. Different implementations of the interface have been created for parallel computers [WM97] and local-area networks of workstations [MS95a]. The first steps toward a wide-area network implementation have also been made. In each case, the assumptions regarding the speed and reliability of the underlying network require a different approach. Here the focus is on providing an abstract machine definition of the LEMMA interface suitable for a reliable local-area network. However, the resulting definition can readily be extended to cover a range of networking scenarios.

The LEMMA interface is based on the Distributed Shared Memory (DSM) model [NL91]. In this model, the memory of the distributed system is treated as a single globally-addressable object. LEMMA statically partitions the address space into a sequence of contiguous semi-spaces, where each semi-space is managed by a different machine (Figure 8.2). Each machine is responsible for the allocation and garbage-collection of the memory in its semi-space.

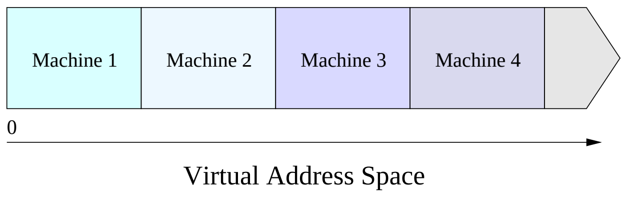


Figure 8.2: *Distributed Shared Memory.*

In a closely-coupled, e.g. parallel or Symmetric Multi-Processing (SMP) implementation, where communication costs are low, memory accesses in the global address space are performed directly. However, in a loosely-coupled, e.g. workstation or Non-Uniform Memory Architecture (NUMA) implementation, the overheads associated with distributed memory access necessitate a form of caching mechanism. The solution adopted by the workstation implementation of LEMMA uses the fact that a typical workstation has a very large virtual address space, but only uses a fraction of it for its real memory. Thus, the virtual address space on each machine is used to represent the entire distributed address space. When a machine wishes to access some external data, it simply addresses its own virtual address space at the required location. The page-faulting mechanism of the operating system is extended to fetch the data from the remote machine, as illustrated in Figure 8.3. Further accesses to the data will simply use the local copy. This operation can be implemented very efficiently on a typical UNIX workstation.

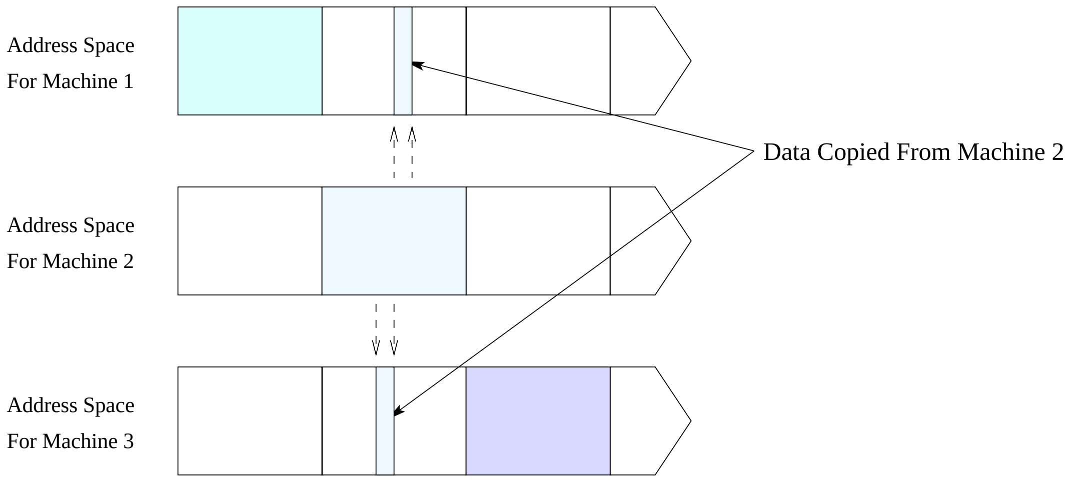

Address Space For Machine 1
Address Space For Machine 2
Address Space For Machine 3
Data Copied From Machine 2

Figure 8.3: *Distributed Memory Access with Caching.*

It is clear that some form of coherency protocol is also required. The original data may be updated, invalidating any copies on other machines. A number of schemes for ensuring coherency are detailed in [SM94]. One significant property of typical functional programs is that most of the data values are immutable. Consequently, LEMMA distinguishes between mutables and immutables and only performs coherency checks on the mutables. For brevity, only the caching of immutables is formalised here. Mutables are simply accessed directly, bypassing the caching mechanism. We consider this acceptable for well-constructed ML programs, which typically avoid using mutable reference values.

8.2 Distributed Abstract Machine

Ideally, we would like our model of distributed evaluation to be an extension of the abstract machine presented in Chapter 5. In order that we may proceed in this direction, we must solve a number of interesting problems. The full language includes primitives for providing communication and concurrency. However, it is far from obvious how these should be expressed in the abstract machine semantics. An examination of the literature on concurrency in structured operational semantics reveals that a 'single-step' (atomic action) approach is typically used to express the computation. Concurrency is then introduced by interleaving the atomic actions of the processes. Unfortunately, this approach is directly in contrast with our relational abstract machine style, where the intermediate steps are hidden and only the final state of the computation is of interest.

At first glance, it appears that the semantics must be rewritten in the single-step style before the concurrent extensions can be expressed. This is undesirable for a number of reasons. The abstract machine model provides an ideal setting for expressing memory management behaviour in a manner that approximates an actual implementation. Furthermore, the single-step approach requires a large number of rules for even the simplest of operations. A further problem can occur when some of the intermediate states do not correspond directly to programs in the language. Techniques exist for automatically performing this transformation in a number of cases [Ber91]. However, the transformed semantics is invariably less clear than the original, and this makes reasoning about the extended language more arduous.

Fortunately, there is an alternative technique inspired by the relational-style semantics of concurrency presented in [Ler92] and [Mit94]. Consider a transition of the form: $(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ E) \Rightarrow (H_2,\ \Delta_2,\ XS_2,\ RS_2)$, expressing the evaluation of an expression E with machine state $(H_1,\ \Delta_1,\ XS_1,\ RS_1)$, and resulting in the final machine state $(H_2,\ \Delta_2,\ XS_2,\ RS_2)$. In the full language, this treatment is no longer acceptable as the evaluation of E may require some external interaction before it can yield the result. Rather than expressing all of the intermediate computations, which are of little interest, we simply present the interactions in the form of an event trace. The trace describes all of the interactions, and the order in which they occur. This leads us to an definition of the form: $(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ E) \overset{t}{\Rightarrow} (H_2,\ \Delta_2,\ XS_2,\ RS_2)$, where t is a trace which records the external interactions required to produce the final result.

$$\text{Trace} \quad t \quad ::= \quad \epsilon \quad | \quad t_1 \; ; \; t_2 \quad | \quad t_1 \parallel t_2 \quad | \quad cn \, ! \, m \quad | \quad cn \, ? \, m$$

Figure 8.4: *Computation and Communication Traces.*

For the $\mathcal{M}\Lambda$ language, the definition of traces in Figure 8.4 is sufficient. A trace may be empty ϵ if no external interactions occur. The sequential evaluation of two expressions leads to a trace of the form $t_1 \; ; \; t_2$. Similarly, two expressions evaluated in parallel produce a trace of the form $t_1 \parallel t_2$. A message m sent over a channel cn yields a trace of the form $cn \; ! \; m$, and a message received on the channel gives the trace $cn \; ? \; m$. Figure 8.5 contains the reduction rules for traces. It should always be the case that the communication traces reduce to ϵ for a complete program, or an error (e.g. deadlock) will have occurred.

$$\epsilon \; ; \; t \quad = \quad t \qquad\qquad t \; ; \; \epsilon \quad = \quad t \tag{8.1}$$

$$\epsilon \parallel t \quad = \quad t \tag{8.2}$$

$$t_1 \parallel t_2 \quad = \quad t_2 \parallel t_1 \tag{8.3}$$

$$(cn \, ! \, m) \parallel (cn \, ? \, m) \quad = \quad \epsilon \tag{8.4}$$

$$(cn \, ! \, m \; ; \; t_1) \parallel (cn \, ? \, m \; ; \; t_2) \quad = \quad t_1 \parallel t_2 \tag{8.5}$$

Figure 8.5: *Trace Reduction Rules.*

The syntax of the distributed abstract machine is defined in Figure 8.6. In order to remain consistent with the LEMMA interface definition, the machine contains two levels of abstraction: *processes* and *threads*. The sequential abstract machine defined in Chapter 5 effectively corresponds to the evaluation of a single thread on a machine containing a single process. In a distributed system there are a fixed number k of concurrently executing processes Π each containing a local heap H, a cache Hc, and a multi-set of threads $\overline{\overline{\overline{T}}}$. It is assumed that these processes are physically distributed, for example, across a network of k machines. The multi-set of threads associated with each process also execute concurrently. However, unlike processes, the multi-set of threads may dynamically grow or shrink. The multi-set may also be empty. Each thread contains a local environment Δ, an exception stack XS, and a result stack RS. The threads share the heap and cache of the parent process. The remainder of the machine is identical to the sequential abstract machine, except that we now allow types and values to be stored directly on the result stack.

Machine State	M	$::=$	$\overline{\Pi}^{k}$
Process	Π	$::=$	$(H,\ Hc,\ \overline{\overline{\mathcal{T}}})$
Thread	$\mathcal{T}$	$::=$	$(\Delta,\ XS,\ RS)$
Trace	t	$::=$	$\epsilon \mid t_1\ ;\ t_2 \mid t_1 \parallel t_2 \mid cn\,!\,m \mid cn\,?\,m$
Message	m	$::=$	$(p) \mid (l,\ p) \mid (\Delta)$
Heap	H	$::=$	$(TH,\ VH)$
Cache	Hc	$::=$	H
Pointer	p		
Type Heap	TH	$::=$	$p \overset{map}{\mapsto} ty$
Heap Types	ty	$::=$	tn
		$\mid$	$tn(p_2)$
		$\mid$	$\overline{p}^{k}$
		$\mid$	$p_1 \rightarrow p_2$
		$\mid$	α
		$\mid$	$\langle\!\langle TE,\ p \rangle\!\rangle$
Location	l		
Value Heap	VH	$::=$	$l \overset{map}{\mapsto} val$
Heap Values	val	$::=$	$scon$
		$\mid$	con
		$\mid$	$con(l)$
		$\mid$	$\overline{l}^{k}$
		$\mid$	$\langle\!\langle \Delta,\ \overline{x}^{k},\ E \rangle\!\rangle$
		$\mid$	Ω
Environment	Δ	$::=$	$(TE,\ VE,\ DE,\ CE)$
Type Variables	TE	$::=$	$\alpha \overset{map}{\mapsto} p$
Lambda Variables	VE	$::=$	$x \overset{map}{\mapsto} (l,\ p)$
Datatype Names	DE	$::=$	$tn \overset{map}{\mapsto} p$
Constructors	CE	$::=$	$con \overset{map}{\mapsto} p$
Exception Stack	XS	$::=$	$()$
		$\mid$	$(l,\ p) \cdot XS$
Result Stack	RS	$::=$	$()$
		$\mid$	$p \cdot RS$
		$\mid$	$(l,\ p) \cdot RS$
		$\mid$	$\Delta \cdot RS$
		$\mid$	$ty \cdot RS$
		$\mid$	$(val,\ ty) \cdot RS$

Figure 8.6: *Distributed Abstract Machine Syntax.*

8.2.1 Machine Evaluation

Given the syntax of the distributed abstract machine, we are left with the task of defining the evaluation of the machine. In the sequential case, we defined relations between successive machine states. However, we are now in the situation where the machine state is defined by processes which, in an actual implementation, will be physically distributed across a number of machines. If we define a sequence of relations over the entire distributed state, we require each process, and hence each machine, to execute synchronously in lock-step. This approach has been used successfully in the definition and implementation of concurrent systems on closely-coupled machines. However, it is entirely inappropriate for an implementation on a loosely-coupled network, where each process will execute at a different rate, and communications may be subject to delays. Imposing synchronisation between processes in this environment would almost certainly lead to an unacceptable performance penalty.

In essence, we require processes and threads to be evaluated asynchronously with respect to one another. The only point at which synchronisation is required is during communication, which is achieved through the use of blocking channels. This leads to an approach whereby each thread is defined in isolation. However, as we have a distributed shared memory model, we require each thread to have direct access to the heap of any other process. We therefore define relations of the form: $(\Pi_1^1,\ \ldots,\ (H_1^i,\ Hc_1^i,\ \{(\Delta_1^n,\ XS_1^n,\ RS_1^n,\ E)\} \uplus \overline{\overline{\overline{\mathcal{T}_1}}}^{\,i}),\ \ldots,\ \Pi_1^k) \overset{t}{\Rightarrow} (\Pi_2^1,\ \ldots,\ (H_2^i,\ Hc_2^i,\ \{(\Delta_2^n,\ XS_2^n,\ RS_2^n)\} \uplus \overline{\overline{\overline{\mathcal{T}_2}}}^{\,i}),\ \ldots,\ \Pi_2^k)$. Each such relation describes the evaluation of a single thread on a single process of the abstract machine. However, the thread has full access to its parent process and all the other processes in the machine. The evaluation trace t contains the interactions for this thread only.

The evaluation of a thread may occur in parallel with the evaluation of any other thread in the machine. The evaluation of the processes and the threads in the system are expressed by the rules in Figure 8.7. These rules establish the fact that processes are evaluated in parallel (Rule 8.6) and that threads are also executed in parallel (Rule 8.7). Note the parallel composition of the traces generated by each process. The execution of the threads is defined inductively; the base case, an empty multi-set of threads, is defined by Rule 8.8. The actual ordering of the evaluation of the threads and processes is left to the non-determinism inherent in these rules. In an actual implementation it would be necessary to implement some manner of scheduling to prevent starvation. However, for our purposes this definition is sufficient.

$$\frac{\Pi_1^1 \overset{t_1}{\Longrightarrow} \Pi_2^1 \quad \cdots \quad \Pi_1^k \overset{t_k}{\Longrightarrow} \Pi_2^k}{(\Pi_1^1,\ \ldots,\ \Pi_1^k) \overset{t_1\|\cdots\|t_k}{\Longrightarrow} (\Pi_2^1,\ \ldots,\ \Pi_2^k)} \tag{8.6}$$

$$\frac{\mathcal{T}_1 \overset{t_1}{\Longrightarrow} \mathcal{T}_3 \qquad \overline{\overline{\mathcal{T}_2}} \overset{t_2}{\Longrightarrow} \overline{\overline{\mathcal{T}_4}}}{\{\mathcal{T}_1\} \uplus \overline{\overline{\mathcal{T}_2}} \overset{t_1\|t_2}{\Longrightarrow} \{\mathcal{T}_3\} \uplus \overline{\overline{\mathcal{T}_4}}} \tag{8.7}$$

$$\frac{}{\emptyset \overset{\epsilon}{\Longrightarrow} \emptyset} \tag{8.8}$$

Figure 8.7: *Concurrent Evaluation of Processes and Threads.*

The initial state of the distributed abstract machine is illustrated in Figure 8.8. The initial heap H^1 and environment Δ^1 are defined in terms of components of the sequential abstract machine, indicated with a subscript e.g. VH_s. The only addition to the heap and environment of the distributed abstract machine is the channel datatype t_chan, used for inter-thread communication. The polymorphic constructor c_chan, which is used to create new channels, has the type $\forall\alpha.\ \alpha \rightarrow \text{t\_chan}(\alpha)$. The type variable α is referenced by the pointer p_{13}.

$$\begin{array}{lcl}
M & = & (\Pi^1,\ (\emptyset,\ \emptyset,\ \emptyset)^2,\ \ldots,\ (\emptyset,\ \emptyset,\ \emptyset)^k) \\[1em]
\Pi^1 & = & (H^1,\ \emptyset,\ \overline{\overline{\mathcal{T}^1}}) \\[1em]
\overline{\overline{\mathcal{T}^1}} & = & \{(\Delta^1,\ (),\ (),\ P)\} \\[1em]
H^1 & = & (TH^1,\ VH_s) \\
TH^1 & = & TH_s \cup \left\{ \begin{array}{l} p_{22} \mapsto \text{t\_chan}(p_{13}),\ p_{23} \mapsto p_{13} \rightarrow p_{22}, \\ p_{24} \mapsto \langle\!\langle \{\alpha \mapsto p_{13}\},\ p_{23} \rangle\!\rangle \end{array} \right\} \\[1em]
\Delta^1 & = & (\emptyset,\ VE_s,\ DE_s,\ CE^1) \\
CE^1 & = & CE_s \cup \{\text{c\_chan} \mapsto p_{24}\}
\end{array}$$

Figure 8.8: *Initial Machine State.*

Evaluation of the distributed abstract machine begins with a single process Π^1, containing the initial heap H^1, and a single thread $(\Delta^1,\ (),\ (),\ P)$. This thread contains the initial environment Δ^1 and the program P. The remaining processes $\Pi^2\ \cdots\ \Pi^k$ remain idle, with an empty heap and multi-set of threads, until a remote thread is created by the program. The evaluation of a program is defined in Figure 8.9. Evaluation terminates when the evaluation of the initial thread has completed. The final result is returned on the result stack of the thread RS_2. The programmer must ensure that the initial thread does not terminate before all of the other threads have terminated. However, this is easily achieved through the use of blocking communication channels. As before, we also have the possibility of non-termination of the abstract machine, or early termination due to an uncaught exception by any thread.

$$\frac{\begin{array}{l} ((H_1,\ Hc_1,\ \{(\Delta_1,\ XS_1,\ RS_1,\ \overline{\overline{D}})\}),\ \Pi_1^2,\ \ldots,\ \Pi_1^k) \overset{t_1}{\Rightarrow} \\ \quad ((H_2,\ Hc_1,\ \{(\Delta_2,\ XS_1,\ RS_1)\}),\ \Pi_1^2,\ \ldots,\ \Pi_1^k) \\ ((H_2,\ Hc_1,\ \{(\Delta_2,\ XS_1,\ RS_1,\ \overline{\overline{X}})\}),\ \Pi_1^2,\ \ldots,\ \Pi_1^k) \overset{t_2}{\Rightarrow} \\ \quad ((H_3,\ Hc_1,\ \{(\Delta_3,\ XS_1,\ RS_1)\}),\ \Pi_1^2,\ \ldots,\ \Pi_1^k) \\ ((H_3,\ Hc_1,\ \{(\Delta_3,\ XS_1,\ RS_1,\ E)\}),\ \Pi_1^2,\ \ldots,\ \Pi_1^k) \overset{t_3}{\Rightarrow} \\ \quad ((H_4,\ Hc_2,\ \{(\Delta_4,\ XS_2,\ RS_2)\}),\ \Pi_2^2,\ \ldots,\ \Pi_2^k) \end{array}}{\begin{array}{l} ((H_1,\ Hc_1,\ \{(\Delta_1,\ XS_1,\ RS_1,\ (\overline{\overline{D}},\ \overline{\overline{X}},\ E))\}),\ \Pi_1^2,\ \ldots,\ \Pi_1^k) \xRightarrow{t_1\ ;\ t_2\ ;\ t_3} \\ \quad ((H_4,\ Hc_2,\ \{(\Delta_4,\ XS_2,\ RS_2)\}),\ \Pi_2^2,\ \ldots,\ \Pi_2^k) \end{array}} \tag{8.9}$$

Figure 8.9: *Program Evaluation.*

There are fundamentally two phases to the evaluation of a program. The first phase is the evaluation of the datatypes $\overline{\overline{D}}$ and exceptions $\overline{\overline{X}}$, and the second phase is the evaluation of the expression E. The first phase is performed sequentially, while the second phase is performed in a distributed manner. This is ensured by the trace $t_1\ ;\ t_2\ ;\ t_3$ in the conclusion of the rule. Note that the processes $\Pi_1^2,\ \cdots,\ \Pi_1^k$ remain unchanged until the expression is evaluated. The datatypes and exceptions are entered into the environment of the initial thread, and the heap of process Π^1. Additional threads are created by constructing closures. Thus, the initial environment Δ_1, together with the datatype and exception declarations are propagated between threads in the environment of these closures. For this reason, the datatype and exception declarations must be evaluated before any threads can be created. Hence, the sequential nature of the first phase of evaluation.

The relations for the datatypes and exceptions are defined below. The rules are very similar to their equivalents in the sequential abstract machine. The datatypes and exceptions are processed sequentially within the initial thread. Although it may appear that they could be processed in parallel, the only way which we can do this is by creating a separate thread for each datatype. This would result in the unacceptable situation where each datatype is contained in a different environment.

$$\frac{}{\begin{array}{l}((H^1,\ Hc^1,\ \{(\Delta,\ XS,\ RS,\ \emptyset)\}),\ \ldots,\ \Pi_1^k) \overset{\epsilon}{\Rightarrow} \\ \quad ((H^1,\ Hc^1,\ \{(\Delta,\ XS,\ RS)\}),\ \ldots,\ \Pi_1^k)\end{array}} \tag{8.10}$$

Comment: (Rule 8.10) An empty set of datatypes or exceptions results in no change to the abstract machine state, hence the ϵ trace.

$$\frac{\begin{array}{l}((H_1^1,\ Hc^1,\ \{(\Delta_1,\ XS,\ RS,\ \{D^1\})\}),\ \ldots,\ \Pi_1^k) \overset{t^1}{\Rightarrow} \\ \quad ((H_2^1,\ Hc^1,\ \{(\Delta_2,\ XS,\ RS)\}),\ \ldots,\ \Pi_1^k) \cdots \\ \quad\quad ((H_k^1,\ Hc^1,\ \{(\Delta_k,\ XS,\ RS,\ \{D^k\})\}),\ \ldots,\ \Pi_1^k) \overset{t^k}{\Rightarrow} \\ \quad\quad\quad ((H_{k+1}^1,\ Hc^1,\ \{(\Delta_{k+1},\ XS,\ RS)\}),\ \ldots,\ \Pi_1^k)\end{array}}{\begin{array}{l}((H_1^1,\ Hc^1,\ \{(\Delta_1,\ XS,\ RS,\ \{D^1,\ \ldots,\ D^k\})\}),\ \ldots,\ \Pi_1^k) \xRightarrow{t^1\ ;\ \cdots\ ;\ t^k} \\ \quad ((H_{k_1}^1,\ Hc^1,\ \{(\Delta_{k+1},\ XS,\ RS)\}),\ \ldots,\ \Pi_1^k)\end{array}} \tag{8.11}$$

Comment: (Rule 8.11) A set of datatype declarations are processed sequentially by the abstract machine, as shown by the resulting trace.

$$\frac{\begin{array}{l}H_1^1 \uparrow p_1 \\ ((H_1^1[p_1\ \mapsto\ tn],\ Hc^1,\ \{(\Delta_1[tn\ \mapsto\ p_1],\ XS,\ RS_1, \\ \quad \tau\_alloc(ctype(C^1,\ tn)))\}),\ \ldots,\ \Pi_1^k) \overset{t^1}{\Rightarrow} \\ \quad\quad ((H_2^1,\ Hc^1,\ \{(\Delta_2,\ XS,\ RS_2)\}),\ \ldots,\ \Pi_1^k)\ \cdots \\ \quad\quad\quad ((H_k^1,\ Hc^1,\ \{(\Delta_k,\ XS,\ RS_k,\ \tau\_alloc(ctype(C^k,\ tn)))\}),\ \ldots,\ \Pi_1^k) \overset{t^k}{\Rightarrow} \\ \quad\quad\quad\quad ((H_{k+1}^1,\ Hc^1,\ \{(\Delta_{k+1},\ XS,\ p_2^k \cdots p_2^1 \cdot RS_1)\}),\ \ldots,\ \Pi_1^k)\end{array}}{\begin{array}{l}((H_1^1,\ Hc^1,\ \{(\Delta_1,\ XS,\ RS_1,\ \textbf{datatype}\ tn\ \textbf{of}\ \{C^1,\ \ldots,\ C^k\})\}),\ \ldots,\ \Pi_1^k) \\ \quad \xRightarrow{t^1\ ;\ \cdots\ ;\ t^k} \\ \quad\quad ((H_{k+1}^1,\ Hc^1,\ \{(\Delta_{k+1}[cname(C^1)\ \mapsto\ p_2^1,\ \ldots,\ cname(C^k)\ \mapsto\ p_2^k], \\ \quad\quad\quad XS,\ RS_1)\}),\ \ldots,\ \Pi_1^k)\end{array}} \tag{8.12}$$

Comment: (Rule 8.12) The constructors of a datatype are also processed sequentially. Note that the heap of the parent process of the thread is accessed directly. The τ_alloc function allocates a type on the heap and will be defined later in this chapter. The *ctype* and *cname* functions have exactly the same definition as in Chapter 5, and are repeated below for convenience:

$$\begin{array}{lcl@{\qquad}lcl}
ctype(con,\ tn) & = & tn & cname(con) & = & con \\
ctype((con,\ \tau),\ tn) & = & \tau \rightarrow tn & cname((con,\ \tau)) & = & con \\
ctype(con,\ (\overline{\alpha}^k,\ tn)) & = & \forall\, \overline{\alpha}^k.\ tn(\overline{\alpha}^k) & & & \\
ctype((con,\ \tau),\ (\overline{\alpha}^k,\ tn)) & = & \forall\, \overline{\alpha}^k.\ \tau \rightarrow tn(\overline{\alpha}^k) & & &
\end{array}$$

$$\frac{\begin{array}{l} H_1^1 \uparrow p_1 \\ ((H_1^1[p_1 \mapsto tn],\ Hc^1,\ \{(\Delta_1[tn \mapsto p_1],\ XS,\ RS_1, \\ \quad \sigma\_alloc(ctype(C^1,\ (\overline{\alpha}^k,\ tn))))\}),\ \ldots,\ \Pi_1^k) \overset{t^1}{\Rightarrow} \\ \quad\quad ((H_2^1,\ Hc^1,\ \{(\Delta_2,\ XS,\ RS_2)\}),\ \ldots,\ \Pi_1^k) \cdots \\ \quad\quad\quad ((H_k^1,\ Hc^1,\ \{(\Delta_k,\ XS,\ RS_k, \\ \quad\quad\quad\quad \sigma\_alloc(ctype(C^k,\ (\overline{\alpha}^k,\ tn))))\}),\ \ldots,\ \Pi_1^k) \overset{t^k}{\Rightarrow} \\ \quad\quad\quad\quad\quad ((H_{k+1}^1,\ Hc^1,\ \{(\Delta_{k+1},\ XS,\ p_1^k \cdots p_1^1 \cdot RS_1)\}),\ \ldots,\ \Pi_1^k) \end{array}}{\begin{array}{l} ((H_1^1,\ Hc^1,\ \{(\Delta_1,\ XS,\ RS_1,\ \mathbf{datatype}\ (\overline{\alpha}^k, tn)\ \mathbf{of}\ \{C^1,\ \ldots,\ C^k\})\}),\ldots,\ \Pi_1^k) \\ \quad \xRightarrow{t^1\ ;\ \cdots\ ;\ t^k} \\ \quad\quad ((H_{k+1}^1,\ Hc^1,\ \{(\Delta_{k+1}[cname(C^1) \mapsto p_1^1,\ \ldots,\ cname(C^k) \mapsto p_1^k], \\ \quad\quad\quad XS,\ RS_1)\}),\ \ldots,\ \Pi_1^k) \end{array}} \tag{8.13}$$

Comment: (Rule 8.13) The rule for polymorphic datatypes is essentially the same as for monomorphic datatypes. The rule proceeds by allocating the datatype name *tn* on the heap, and entering the corresponding pointer p_1 into the environment. The type schemes for the constructors are then allocated sequentially on the heap using the *σ_alloc* function, which will also be defined later in this chapter. Finally, the environment is updated with a mapping from each constructor name to its heap allocated type.

$$\frac{\begin{array}{l} ((H_1^1,\ Hc^1,\ \{(\Delta_1,\ XS,\ RS_1,\ \tau\_alloc(ctype(C^1,\ \mathrm{t\_exn})))\}),\ \ldots,\ \Pi_1^k) \overset{t^1}{\Rightarrow} \\ \quad ((H_2^1,\ Hc^1,\ \{(\Delta_2,\ XS,\ RS_2)\}),\ \ldots,\ \Pi_1^k) \cdots \\ \quad ((H_k^1,\ Hc^1,\ \{(\Delta_k,\ XS,\ RS_k,\ \tau\_alloc(ctype(C^k,\ \mathrm{t\_exn})))\}),\ \ldots,\ \Pi_1^k) \overset{t^k}{\Rightarrow} \\ \quad\quad ((H_{k+1}^1,\ Hc^1,\ \{(\Delta_{k+1},\ XS,\ p^k \cdots p^1 \cdot RS_1)\}),\ \ldots,\ \Pi_1^k) \end{array}}{\begin{array}{l} ((H_1^1,\ Hc^1,\ \{(\Delta_1,\ XS,\ RS_1, \{\mathbf{exception}\, C^1, \ldots, \mathbf{exception}\, C^k\})\}),\ \ldots,\ \Pi_1^k) \\ \quad \xRightarrow{t^1\ ;\ \cdots\ ;\ t^k} \\ \quad\quad ((H_{k+1}^1,\ Hc^1,\ \{(\Delta_{k+1}[cname(C^1) \mapsto p^1,\ \ldots,\ cname(C^k) \mapsto p^k], \\ \quad\quad\quad XS,\ RS_1)\}),\ \ldots,\ \Pi_1^k) \end{array}} \tag{8.14}$$

Comment: (Rule 8.14) Recall that exceptions are treated as constructors of a reserved datatype t_exn, and polymorphic exceptions are forbidden. Therefore, this rule is essentially a specialised variant of Rule 8.12 for a datatype named t_exn.

8.2.2 Memory Access and Caching

The distributed abstract machine uses the LEMMA model of distributed shared memory. This means that any thread can directly access the data contained on the heap of any other process, e.g. $H^i(p) = ty$, where $0 < i \leq k$. However, it was argued in Section 8.1 that direct memory access is impractical in a loosely-coupled system owing to communication overheads. In this section we model the caching of remote memory within the abstract machine. In order for this to be effective, *all* memory accesses in the abstract machine must be performed via a *fetch* function, which we define below. The result of this function is a type ty or pair $(val,\ ty)$ on the result stack.

Each process in the abstract machine has a heap H and a cache Hc. With reference to Figures 8.2 and 8.3, the area of the virtual address space which corresponds to the semi-space managed by a particular machine is represented by the heap. The remainder of the virtual address space is represented by the cache. Clearly, the heap will be tightly populated, while the cache will be sparsely populated. The cache is cleared during garbage collection. The abstract machine does not model virtual memory directly as there is no notion of memory addresses, and the virtual memory system is inherently platform dependent. However, it is worth noting that in an actual implementation, the heap and cache would be mapped onto a virtual address space. This differs from the sequential abstract machine, where it was assumed that the heap was a direct model of the physical memory.

The $fetch$ function retrieves only a single type or value at a time. This granularity will typically be too fine for an actual implementation due to communication overheads. In the LEMMA interface, a whole *page* of data is fetched on each remote memory access. This is based on the principle of locality: the next location of memory that is required will usually be close to the previous location. Unfortunately we cannot model this here as we have no way of determining if one pointer or location is close to another in the heap. Also, the heap is not divided into pages, as the size of a page is fundamentally a machine-dependent property and is therefore inappropriate for our model.

Fetching and caching of types is defined in Rules 8.15 through 8.18. The $fetch(p)$ function returns the type referenced by the type-pointer p on the result stack RS. The rules utilise a function $num(p)$ which return the number of the process whose heap contains the type-pointer p. There are four rules corresponding to local pointers, cached pointers, remote mutable pointers and remote immutable pointers.

$$\frac{num(p) = i}{\begin{array}{l}(\Pi^1,\ \ldots,\ (H^i,\ Hc^i,\ \{(\Delta^n,\ XS^n,\ RS^n,\ fetch(p))\} \uplus \overline{\overline{\mathcal{T}^i}}),\ \ldots,\ \Pi^k) \overset{\epsilon}{\Rightarrow} \\ \quad (\Pi^1,\ \ldots,\ (H^i,\ Hc^i,\ \{(\Delta^n,\ XS^n,\ H^i(p) \cdot RS^n)\} \uplus \overline{\overline{\mathcal{T}^i}}),\ \ldots,\ \Pi^k)\end{array}} \tag{8.15}$$

Comment: (Rule 8.15) If the pointer p is contained in the heap of the parent process H^i, then the type is obtained directly from the heap and returned on the result stack.

$$\frac{num(p) \neq i \qquad p \in \text{Dom } Hc^i}{\begin{array}{l}(\Pi^1,\ \ldots,\ (H^i,\ Hc^i,\ \{(\Delta^n,\ XS^n,\ RS^n,\ fetch(p))\} \uplus \overline{\overline{\mathcal{T}^i}}),\ \ldots,\ \Pi^k) \overset{\epsilon}{\Rightarrow} \\ \quad (\Pi^1,\ \ldots,\ (H^i,\ Hc^i,\ \{(\Delta^n,\ XS^n,\ Hc^i(p) \cdot RS^n)\} \uplus \overline{\overline{\mathcal{T}^i}}),\ \ldots,\ \Pi^k)\end{array}} \tag{8.16}$$

Comment: (Rule 8.15) If the pointer p is non-local $num(p) \neq i$, but is present in the local cache $p \in$ Dom Hc^i, then the type is obtained directly from the cache and returned on the result stack.

$$\frac{num(p_1) = j \qquad j \neq i \qquad p_1 \neq \text{Dom } Hc^i \qquad H^j(p_1) = \text{t\_ref}(p_2)}{\begin{array}{l}(\Pi^1,\ \ldots,\ (H^i,\ Hc^i,\ \{(\Delta^n,\ XS^n,\ RS^n,\ fetch(p_1))\} \uplus \overline{\overline{\mathcal{T}^i}}),\ \ldots \\ \quad (H^j,\ Hc^j,\ \overline{\overline{\mathcal{T}^j}}),\ \ldots,\ \Pi^k) \overset{\epsilon}{\Rightarrow} \\ \qquad (\Pi^1,\ \ldots,\ (H^i,\ Hc^i,\ \{(\Delta^n,\ XS^n,\ \text{t\_ref}(p_2) \cdot RS^n)\} \uplus \overline{\overline{\mathcal{T}^i}}),\ \ldots \\ \qquad\quad (H^j,\ Hc^j,\ \overline{\overline{\mathcal{T}^j}}),\ \ldots,\ \Pi^k)\end{array}} \tag{8.17}$$

Comment: (Rule 8.15) If the pointer p_1 is non-local and is not in the cache, then the remote type is retrieved $H^j(p_1)$. If this type is a reference (i.e. mutable) then the type is simply returned on the result stack.

$$\frac{num(p_1) = j \qquad j \neq i \qquad p_1 \neq \text{Dom } Hc^i \qquad H^j(p_1) = ty \qquad ty \neq \text{t\_ref}(p_2)}{\begin{array}{l}(\Pi^1,\ \ldots,\ (H^i,\ Hc^i,\ \{(\Delta^n,\ XS^n,\ RS^n,\ fetch(p_1))\} \uplus \overline{\overline{\mathcal{T}^i}}),\ \ldots \\ \quad (H^j,\ Hc^j,\ \overline{\overline{\mathcal{T}^j}}),\ \ldots,\ \Pi^k) \overset{\epsilon}{\Rightarrow} \\ \qquad (\Pi^1,\ \ldots,\ (H^i,\ Hc^i[p_1 \ \mapsto\ ty],\ \{(\Delta^n,\ XS^n,\ ty \cdot RS^n)\} \uplus \overline{\overline{\mathcal{T}^i}}),\ \ldots \\ \qquad\quad (H^j,\ Hc^j,\ \overline{\overline{\mathcal{T}^j}}),\ \ldots,\ \Pi^k)\end{array}} \tag{8.18}$$

Comment: (Rule 8.18) If the pointer p_1 is non-local, is not in the cache, and does not reference a mutable type, then the type is stored in the local cache $Hc^i[p_1 \ \mapsto\ ty]$ and returned on the result stack.

The retrieval and caching of values is defined by Rule 8.19 through 8.22. The $fetch(l,\ p)$ function returns a pair of a type and a value $(val,\ ty)$ on the result stack. As with the types, there four rules corresponding to local, cached, mutable, and immutable values. The rules are slightly more complex as the type must first be obtained via a call to $fetch(p)$ before the value is retrieved. We cannot guarantee that a value will be in the same heap as its corresponding type. Note that the type is used to determine if value is mutable or immutable.

$$\frac{\begin{array}{l} num(l) = i \\ (\Pi_1^1,\ \ldots,\ (H_1^i,\ Hc_1^i,\ \{(\Delta^n,\ XS^n,\ RS^n,\ fetch(p))\} \uplus \overline{\overline{\overline{\mathcal{T}_1^i}}}),\ \ldots,\ \Pi_1^k) \overset{t}{\Rightarrow} \\ \quad (\Pi_2^1,\ \ldots,\ (H_2^i,\ Hc_2^i,\ \{(\Delta^n,\ XS^n,\ ty \cdot RS^n)\} \uplus \overline{\overline{\overline{\mathcal{T}_2^i}}}),\ \ldots,\ \Pi_2^k) \end{array}}{\begin{array}{l} (\Pi_1^1,\ \ldots,\ (H_1^i,\ Hc_1^i,\ \{(\Delta^n,\ XS^n,\ RS^n,\ fetch(l,\ p))\} \uplus \overline{\overline{\overline{\mathcal{T}_1^i}}}),\ \ldots,\ \Pi_1^k) \overset{t}{\Rightarrow} \\ \quad (\Pi_2^1,\ \ldots,\ (H_2^i,\ Hc_2^i,\ \{(\Delta^n,\ XS^n,\ (H_2^i(l),\ ty) \cdot RS^n)\} \uplus \overline{\overline{\overline{\mathcal{T}_2^i}}}),\ \ldots,\ \Pi_2^k) \end{array}} \tag{8.19}$$

$$\frac{\begin{array}{l} num(l) \neq i \qquad l \in \text{Dom } Hc_1^i \\ (\Pi_1^1,\ \ldots,\ (H_1^i,\ Hc_1^i,\ \{(\Delta^n,\ XS^n,\ RS^n,\ fetch(p))\} \uplus \overline{\overline{\overline{\mathcal{T}_1^i}}}),\ \ldots,\ \Pi_1^k) \overset{t}{\Rightarrow} \\ \quad (\Pi_2^1,\ \ldots,\ (H_2^i,\ Hc_2^i,\ \{(\Delta^n,\ XS^n,\ ty \cdot RS^n)\} \uplus \overline{\overline{\overline{\mathcal{T}_2^i}}}),\ \ldots,\ \Pi_2^k) \end{array}}{\begin{array}{l} (\Pi_1^1,\ \ldots,\ (H_1^i,\ Hc_1^i,\ \{(\Delta^n,\ XS^n,\ RS^n,\ fetch(l,\ p))\} \uplus \overline{\overline{\overline{\mathcal{T}_1^i}}}),\ \ldots,\ \Pi_1^k) \overset{t}{\Rightarrow} \\ \quad (\Pi_2^1,\ \ldots,\ (H_2^i,\ Hc_2^i,\ \{(\Delta^n,\ XS^n,\ (Hc_2^i(l),\ ty) \cdot RS^n)\} \uplus \overline{\overline{\overline{\mathcal{T}_2^i}}}),\ \ldots,\ \Pi_2^k) \end{array}} \tag{8.20}$$

$$\frac{\begin{array}{l} num(l) = j \qquad j \neq i \qquad l \notin \text{Dom } Hc_1^i \\ (\Pi_1^1,\ \ldots,\ (H_1^i,\ Hc_1^i,\ \{(\Delta^n,\ XS^n,\ RS^n,\ fetch(p_1))\} \uplus \overline{\overline{\overline{\mathcal{T}_1^i}}}),\ \ldots,\ \Pi_1^k) \overset{t}{\Rightarrow} \\ \quad (\Pi_2^1,\ \ldots,\ (H_2^i,\ Hc_2^i,\ \{(\Delta^n,\ XS^n,\ ty \cdot RS^n)\} \uplus \overline{\overline{\overline{\mathcal{T}_2^i}}}),\ \ldots \\ \qquad (H_2^j,\ Hc_2^j,\ \overline{\overline{\overline{\mathcal{T}_2^j}}}),\ \ldots,\ \Pi_2^k) \\ ty = \text{t\_ref}(p_2) \end{array}}{\begin{array}{l} (\Pi_1^1,\ \ldots,\ (H_1^i,\ Hc_1^i,\ \{(\Delta^n,\ XS^n,\ RS^n,\ fetch(l,\ p_1))\} \uplus \overline{\overline{\overline{\mathcal{T}_1^i}}}),\ \ldots,\ \Pi_1^k) \overset{t}{\Rightarrow} \\ \quad (\Pi_2^1,\ \ldots,\ (H_2^i,\ Hc_2^i,\ \{(\Delta^n,\ XS^n,\ (H_2^j(l),\ ty) \cdot RS^n)\} \uplus \overline{\overline{\overline{\mathcal{T}_2^i}}}),\ \ldots \\ \qquad (H_2^j,\ Hc_2^j,\ \overline{\overline{\overline{\mathcal{T}_2^j}}}),\ \ldots,\ \Pi_2^k) \end{array}} \tag{8.21}$$

$$\frac{\begin{array}{l} num(l) = j \qquad j \neq i \qquad l \notin \text{Dom } Hc_1^i \\ (\Pi_1^1,\ \ldots,\ (H_1^i,\ Hc_1^i,\ \{(\Delta^n,\ XS^n,\ RS^n,\ fetch(p_1))\} \uplus \overline{\overline{\overline{\mathcal{T}_1^i}}}),\ \ldots,\ \Pi_1^k) \overset{t}{\Rightarrow} \\ \quad (\Pi_2^1,\ \ldots,\ (H_2^i,\ Hc_2^i,\ \{(\Delta^n,\ XS^n,\ ty \cdot RS^n)\} \uplus \overline{\overline{\overline{\mathcal{T}_2^i}}}),\ \ldots \\ \qquad (H_2^j,\ Hc_2^j,\ \overline{\overline{\overline{\mathcal{T}_2^j}}}),\ \ldots,\ \Pi_2^k) \\ ty \neq \text{t\_ref}(p_2) \qquad H_2^j(l) = val \end{array}}{\begin{array}{l} (\Pi_1^1,\ \ldots,\ (H_1^i,\ Hc_1^i,\ \{(\Delta^n,\ XS^n,\ RS^n,\ fetch(l,\ p))\} \uplus \overline{\overline{\overline{\mathcal{T}_1^i}}}),\ \ldots,\ \Pi_1^k) \overset{t}{\Rightarrow} \\ \quad (\Pi_2^1,\ \ldots,\ (H_2^i,\ Hc_2^i[l\ \mapsto\ val],\ \{(\Delta^n,\ XS^n,\ (val,\ ty) \cdot RS^n)\} \uplus \overline{\overline{\overline{\mathcal{T}_2^i}}}),\ \ldots \\ \qquad (H_2^j,\ Hc_2^j,\ \overline{\overline{\overline{\mathcal{T}_2^j}}}),\ \ldots,\ \Pi_2^k) \end{array}} \tag{8.22}$$

8.2.3 Memory Allocation

Memory allocation in the distributed abstract machine is more straightforward than memory access. Although a thread can access any heap in the distributed shared memory, it can only perform allocation locally within the heap of the parent process. Therefore, the allocation of a type or value is performed directly as before, e.g. $H^i \uparrow p \; H^i[p \mapsto ty]$, where H^i is the local heap. However, we still require distributed versions of the functions τ_alloc, α_alloc and σ_alloc for converting static types into dynamic types, and the *instance* function for instantiating type schemes.

A large number of the transitions in the distributed abstract machine are essentially the same as their sequential counterparts. Therefore, rather than presenting all of the relations in full, we will simply specify translations between equivalent sequential and distributed relations in the form of *templates*. The first two such templates are shown in Figure 8.10. The first template (8.23) converts a sequential relation into a distributed one, where *fn* is the name of a function, and p is the resulting type pointer. Note that the heap H is translated into the heap of the parent process H^i, while the environment and stacks are translated into those of the thread Δ^n, etc.

The second template (8.24) is used to convert a direct memory access into one which uses the $fetch$ function. These templates are enough to convert all of the type functions τ_alloc, α_alloc, σ_alloc, and *instance*. Note that the templates cannot be applied blindly; some renumbering of the tuples is necessary, and an evaluation trace must be constructed. As an example, the translated α_alloc function is shown in Figure 8.11.

Sequential : $(H_1,\ \Delta_1,\ XS,\ RS,\ fn) \Rightarrow (H_2,\ \Delta_2,\ XS,\ p \cdot RS)$

Distributed :

$$\begin{array}{l}(\Pi^1_1,\ \ldots,\ (H^i_1,\ Hc^i_1,\ \{(\Delta^n_1,\ XS^n,\ RS^n,\ fn)\} \uplus \overline{\overline{\mathcal{T}^i_1}}),\ \ldots,\ \Pi^k_1) \overset{t}{\Rightarrow} \\ \quad (\Pi^1_2,\ \ldots,\ (H^i_2,\ Hc^i_2,\ \{(\Delta^n_2,\ XS^n,\ p \cdot RS^n)\} \uplus \overline{\overline{\mathcal{T}^i_2}}),\ \ldots,\ \Pi^k_2)\end{array} \tag{8.23}$$

Sequential : $H_1(p) = ty$

Distributed :

$$\begin{array}{l}(\Pi^1_1,\ \ldots,\ (H^i_1,\ Hc^i_1,\ \{(\Delta^n_1,\ XS^n_1,\ RS^n_1,\ fetch(p))\} \uplus \overline{\overline{\mathcal{T}^i_1}}),\ \ldots,\ \Pi^k_1) \overset{t}{\Rightarrow} \\ \quad (\Pi^1_2,\ \ldots,\ (H^i_2,\ Hc^i_2,\ \{(\Delta^n_2,\ XS^n_2,\ ty \cdot RS^n_2)\} \uplus \overline{\overline{\mathcal{T}^i_2}}),\ \ldots,\ \Pi^k_2)\end{array} \tag{8.24}$$

Figure 8.10: *Type Function Templates.*

$$\frac{}{\begin{array}{l}(\Pi^1,\ \ldots,\ (H^i,\ Hc^i,\ \{(\Delta^n,\ XS^n,\ RS^n,\ \alpha\_alloc(\tau))\} \uplus \overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi^k) \overset{\epsilon}{\Rightarrow} \\ \quad (\Pi^1,\ \ldots,\ (H^i,\ Hc^i,\ \{(\Delta^n,\ XS^n,\ RS^n)\} \uplus \overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi^k)\end{array}} \tag{8.25}$$

$$\frac{H^i \uparrow (p^1,\ \ldots,\ p^k)}{\begin{array}{l}(\Pi^1,\ \ldots,\ (H^i,\ Hc^i,\ \{(\Delta^n,\ XS^n,\ RS^n,\ \alpha\_alloc(\forall\,\overline{\alpha}^k.\ \tau))\} \uplus \overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi^k) \overset{\epsilon}{\Rightarrow} \\ \quad (\Pi^1,\ \ldots,\ (H^i[p^1 \mapsto \alpha^1,\ \ldots,\ p^k \mapsto \alpha^k],\ Hc^i, \\ \qquad \{(\Delta^n[\alpha^1 \mapsto p^1,\ \ldots,\ \alpha^k \mapsto p^k],\ XS^n,\ RS^n)\} \uplus \overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi^k)\end{array}} \tag{8.26}$$

Figure 8.11: *Distributed α_alloc Function.*

The translated *instance* function is given in Figure 8.12. This rule neatly illustrates the composition of the evaluation traces in the distributed abstract machine. If there are no transitions in the premise of the rule, as in the *α_alloc* function, then the evaluation trace is empty ϵ. If there is one transition in the premise of the rule, as in the *fetch* function (Rule 8.22), then the evaluation trace t of the premise and conclusion are the same. However, if there is more than one transition in the premise of the rule, as in the *instance* function, then the traces are sequentially composed in the conclusion of the rule $t_1\ ;\ t_2$. It is necessary to compose the traces in this manner when performing the template-based translations.

$$\frac{\begin{array}{l}(\Pi_1^1,\ \ldots,\ (H_1^i,\ Hc_1^i,\ \{(\Delta^n,\ XS^n,\ RS^n,\ fetch(p_\sigma))\} \uplus \overline{\overline{\mathcal{T}_1^i}})),\ \ldots,\ \Pi_1^k) \overset{t_1}{\Rightarrow} \\ \quad (\Pi_1^1,\ \ldots,\ (H_2^i,\ Hc_2^i, \\ \qquad \{(\Delta^n,\ XS^n,\ \langle\!\langle\{\alpha^1 \mapsto p_1^1,\ \ldots,\ \alpha^k \mapsto p_1^k\},\ p_2\rangle\!\rangle \cdot RS^n)\} \uplus \overline{\overline{\mathcal{T}_1^i}})),\ \ldots,\ \Pi_1^k) \\ (\Pi_1^1,\ \ldots,\ (H_2^i,\ Hc_2^i,\ \{(\Delta^n,\ XS^n,\ RS^n,\ fetch(p_\tau))\}),\ \ldots,\ \Pi_1^k) \overset{t_2}{\Rightarrow} \\ \quad (\Pi_1^1,\ \ldots,\ (H_3^i,\ Hc_3^i,\ \{(\Delta^n,\ XS^n,\ \overline{p_3}^k \cdot RS^n)\} \uplus \overline{\overline{\mathcal{T}_1^i}})),\ \ldots,\ \Pi_1^k)\end{array}}{\begin{array}{l}(\Pi_1^1,\ \ldots,\ (H_1^i,\ Hc_1^i,\ \{(\Delta^n,\ XS^n,\ RS^n,\ instance(p_\sigma,\ p_\tau))\} \uplus \overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi_1^k) \\ \quad \xRightarrow{t_1\ ;\ t_2} \\ \qquad (\Pi_1^1,\ \ldots,\ (H_3^i[p_4 \mapsto \langle\!\langle\{\alpha^1 \mapsto p_3^1,\ \ldots,\ \alpha^k \mapsto p_3^k\},\ p_2\rangle\!\rangle],\ Hc_3^i, \\ \qquad\quad \{(\Delta^n,\ XS^n,\ p_4 \cdot RS^n)\} \uplus \overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi_1^k)\end{array}} \tag{8.27}$$

Figure 8.12: *Distributed Type Scheme Instantiation.*

8.2.4 Thread Creation and Communication

We can now define operations for thread creation and communication. Our abstract machine definition has been designed to allow these operations to be performed with relative ease. In $\mathcal{M}\Lambda$, a thread is represented by a functions of type t_unit → t_unit. A thread is created by evaluating this function with a **fork** or **rfork** expression. The **fork** expression evaluates a thread on the same process as the thread performing the **fork**, while the **rfork** expression evaluates a thread on a remote process, supplied as an integer argument.

$$\frac{\begin{array}{l}(\Pi^1_1,\ \ldots,\ (H^i_1,\ Hc^i_1,\ \{(\Delta^n_1,\ XS^n_1,\ RS^n_1,\ E)\} \uplus \overline{\overline{\mathcal{T}^i_1}}),\ \ldots,\ \Pi^k_1) \overset{t_1}{\Rrightarrow} \\ \quad (\Pi^1_2,\ \ldots,\ (H^i_2,\ Hc^i_2,\ \{(\Delta^n_2,\ XS^n_2,\ (l_1,\ p_1)\cdot RS^n_2)\} \uplus \overline{\overline{\mathcal{T}^i_2}}),\ \ldots,\ \Pi^k_2) \\ (\Pi^1_2,\ \ldots,\ (H^i_2,\ Hc^i_2,\ \{(\Delta^n_2,\ XS^n_2,\ RS^n_2,\ fetch(l_1,\ p_1))\} \uplus \overline{\overline{\mathcal{T}^i_2}}),\ \ldots,\ \Pi^k_2) \overset{t_2}{\Rrightarrow} \\ \quad (\Pi^1_3,\ \ldots,\ (H^i_3,\ Hc^i_3,\ \{(\Delta^n_3,\ XS^n_3, \\ \qquad (\langle\!\langle \Delta_c,\ (x),\ E_c\rangle\!\rangle,\ ty)\cdot RS^n_3)\} \uplus \overline{\overline{\mathcal{T}^i_3}}),\ \ldots,\ \Pi^k_3)\end{array}}{\begin{array}{l}(\Pi^1_1,\ \ldots,\ (H^i_1,\ Hc^i_1,\ \{(\Delta^n_1,\ XS^n_1,\ RS^n_1,\ \mathbf{fork}\ E)\} \uplus \overline{\overline{\mathcal{T}^i_1}}),\ \ldots,\ \Pi^k_1) \overset{t_1\ ;\ t_2}{\Longrightarrow} \\ \quad (\Pi^1_3,\ \ldots,\ (H^i_3,\ Hc^i_3,\ \{(\Delta^n_3,\ XS^n_3,\ \Delta^n_3(UNIT)\cdot RS^n_3)\} \uplus \\ \qquad \{(\Delta_c,\ \emptyset,\ \emptyset,\ E_c)\} \uplus \overline{\overline{\mathcal{T}^i_3}}),\ \ldots,\ \Pi^k_3)\end{array}} \quad (8.28)$$

Comment: (Rule 8.28) The argument expression E is evaluated to obtain a closure $\langle\!\langle \Delta_c,\ (x),\ E_c\rangle\!\rangle$. This closure is evaluated by creating a thread $(\Delta_c,\ \emptyset,\ \emptyset,\ E_c)$ which is appended to the local multi-set of threads.

$$\frac{\begin{array}{l}(\Pi^1_1,\ \ldots,\ (H^i_1,\ Hc^i_1,\ \{(\Delta^n_1,\ XS^n_1,\ RS^n_1,\ E_1)\} \uplus \overline{\overline{\mathcal{T}^i_1}}),\ \ldots,\ \Pi^k_1) \overset{t_1}{\Rrightarrow} \\ \quad (\Pi^1_2,\ \ldots,\ (H^i_2,\ Hc^i_2,\ \{(\Delta^n_2,\ XS^n_2,\ (l_1,\ p_1)\cdot RS^n_2)\} \uplus \overline{\overline{\mathcal{T}^i_2}}),\ \ldots,\ \Pi^k_2) \\ (\Pi^1_2,\ \ldots,\ (H^i_2,\ Hc^i_2,\ \{(\Delta^n_2,\ XS^n_2,\ RS^n_2,\ fetch(l_1,\ p_1))\} \uplus \overline{\overline{\mathcal{T}^i_2}}),\ \ldots,\ \Pi^k_2) \overset{t_2}{\Rrightarrow} \\ \quad (\Pi^1_3,\ \ldots,\ (H^i_3,\ Hc^i_3,\ \{(\Delta^n_3,\ XS^n_3,\ RS^n_3)\} \uplus \overline{\overline{\mathcal{T}^i_3}}),\ \ldots,\ \Pi^k_3) \\ (\Pi^1_3,\ \ldots,\ (H^i_3,\ Hc^i_3,\ \{(\Delta^n_3,\ XS^n_3,\ RS^n_3,\ E_2)\} \uplus \overline{\overline{\mathcal{T}^i_3}}),\ \ldots,\ \Pi^k_3) \overset{t_3}{\Rrightarrow} \\ \quad (\Pi^1_4,\ \ldots,\ (H^i_4,\ Hc^i_4,\ \{(\Delta^n_4,\ XS^n_4,\ (l_2,\ p_2)\cdot(j,\ ty_1)\cdot RS^n_4)\} \uplus \overline{\overline{\mathcal{T}^i_4}}),\ \ldots,\ \Pi^k_4) \\ (\Pi^1_4,\ \ldots,\ (H^i_4,\ Hc^i_4,\ \{(\Delta^n_4,\ XS^n_4,\ RS^n_4,\ fetch(l_2,\ p_2))\} \uplus \overline{\overline{\mathcal{T}^i_4}}),\ \ldots,\ \Pi^k_4) \overset{t_4}{\Rrightarrow} \\ \quad (\Pi^1_5,\ \ldots,\ (H^i_5,\ Hc^i_5,\ \{(\Delta^n_5,\ XS^n_5,\ (\langle\!\langle \Delta_c,\ (x),\ E_c\rangle\!\rangle,\ ty_2)\cdot RS^n_5)\} \uplus \overline{\overline{\mathcal{T}^i_5}}),\ \ldots \\ \qquad (H^{(j \bmod k)},\ Hc^{(j \bmod k)},\ \overline{\overline{\mathcal{T}^{(j \bmod k)}}}),\ \ldots,\ \Pi^k_5)\end{array}}{\begin{array}{l}(\Pi^1_1,\ \ldots,\ (H^i_1,\ Hc^i_1,\ \{(\Delta^n_1,\ XS^n_1,\ RS^n_1,\ \mathbf{rfork}\ (E_1,\ E_2))\} \uplus \overline{\overline{\mathcal{T}^i_1}}),\ \ldots,\ \Pi^k_1) \\ \quad \overset{t_1\ ;\ t_2\ ;\ t_3\ ;\ t_4}{\Longrightarrow} \\ \qquad (\Pi^1_5,\ \ldots,\ (H^i_5,\ Hc^i_5,\ \{(\Delta^n_5,\ XS^n_5,\ \Delta^n_5(UNIT)\cdot RS^n_5)\} \uplus \overline{\overline{\mathcal{T}^i_5}}),\ \ldots \\ \qquad\quad (H^{(j \bmod k)},\ Hc^{(j \bmod k)},\ \{(\Delta_c,\ \emptyset,\ \emptyset,\ E_c)\} \uplus \overline{\overline{\mathcal{T}^{(j \bmod k)}}}),\ \ldots,\ \Pi^k_5)\end{array}} \quad (8.29)$$

Comment: (Rule 8.29) The expression E_1 is first evaluated to determine the remote process number j. E_2 is then evaluated to provide a closure, which is converted to a thread and appended to the multi-set of threads on process Π^j.

Communication between threads is performed across *blocking* bi-directional channels; both the sending and receiving threads are blocked until communication is complete. Recall from Chapter 2 that channels are represented by instances of the c_chan(α) constructor. Thus, at the abstract machine level, channels are represented by nullary constructor values *con*. For clarity, we will write channel constructor values as *cn*. Messages are communicated by the **send** and **receive** expressions. The messages communicated by these expressions are always pairs $(l,\ p)$ (the message types p and Δ will be used in Chapter 9). The notation for sending and receiving across channels is the same as the communication traces. For example, sending the pair $(l,\ p)$ across channel cn is written $cn\,!\,(l,\ p)$.

$$\frac{\begin{array}{l}(\Pi_1^1,\ \ldots,\ (H_1^i,\ Hc_1^i,\ \{(\Delta_1^n,\ XS_1^n,\ RS_1^n,\ E_1)\} \uplus \overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi_1^k) \overset{t_1}{\Rightarrow} \\ \quad (\Pi_2^1,\ \ldots,\ (H_2^i,\ Hc_2^i,\ \{(\Delta_2^n,\ XS_2^n,\ (l_1,\ p_1)\cdot RS_2^n)\} \uplus \overline{\overline{\mathcal{T}_2^i}}),\ \ldots,\ \Pi_2^k) \\ (\Pi_2^1,\ \ldots,\ (H_2^i,\ Hc_2^i,\ \{(\Delta_2^n,\ XS_2^n,\ RS_2^n,\ fetch(l_1,\ p_1))\} \uplus \overline{\overline{\mathcal{T}_2^i}}),\ \ldots,\ \Pi_2^k) \overset{t_2}{\Rightarrow} \\ \quad (\Pi_3^1,\ \ldots,\ (H_3^i,\ Hc_3^i,\ \{(\Delta_3^n,\ XS_3^n,\ RS_3^n)\} \uplus \overline{\overline{\mathcal{T}_3^i}}),\ \ldots,\ \Pi_3^k) \\ (\Pi_3^1,\ \ldots,\ (H_3^i,\ Hc_3^i,\ \{(\Delta_3^n,\ XS_3^n,\ RS_3^n,\ E_2)\} \uplus \overline{\overline{\mathcal{T}_3^i}}),\ \ldots,\ \Pi_3^k) \overset{t_3}{\Rightarrow} \\ \quad (\Pi_4^1,\ \ldots,\ (H_4^i,\ Hc_4^i,\ \{(\Delta_4^n,\ XS_4^n,\ (l_2,\ p_2)\cdot(cn,\ ty)\cdot RS_4^n)\} \uplus \overline{\overline{\mathcal{T}_4^i}}),\ \ldots,\ \Pi_4^k) \\ cn\,!\,(l_2,\ p_2)\end{array}}{\begin{array}{l}(\Pi_1^1,\ \ldots,\ (H_1^i,\ Hc_1^i,\ \{(\Delta_1^n,\ XS_1^n,\ RS_1^n,\ \mathbf{send}\,(E_1,\ E_2))\} \uplus \overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi_1^k) \\ \quad \xRightarrow{t_1\,;\,t_2\,;\,t_3\,;\,cn\,!\,(l_2,\ p_2)} \\ \qquad (\Pi_4^1,\ \ldots,\ (H_4^i,\ Hc_4^i,\ \{(\Delta_4^n,\ XS_4^n,\ \Delta_4^n(UNIT)\cdot RS_4^n)\} \uplus \overline{\overline{\mathcal{T}_4^i}}),\ \ldots,\ \Pi_4^k)\end{array}} \tag{8.30}$$

Comment: (Rule 8.30) The expression E_1 is evaluated to obtain the channel $(cn,\ ty)$. The expression E_2 is evaluated to form the message $(l_2,\ p_2)$ which is sent along the channel.

$$\frac{\begin{array}{l}(\Pi_1^1,\ \ldots,\ (H_1^i,\ Hc_1^i,\ \{(\Delta_1^n,\ XS_1^n,\ RS_1^n,\ E)\} \uplus \overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi_1^k) \overset{t_1}{\Rightarrow} \\ \quad (\Pi_2^1,\ \ldots,\ (H_2^i,\ Hc_2^i,\ \{(\Delta_2^n,\ XS_2^n,\ (l_1,\ p_1)\cdot RS_2^n)\} \uplus \overline{\overline{\mathcal{T}_2^i}}),\ \ldots,\ \Pi_2^k) \\ (\Pi_2^1,\ \ldots,\ (H_2^i,\ Hc_2^i,\ \{(\Delta_2^n,\ XS_2^n,\ RS_2^n,\ fetch(l_1,\ p_1))\} \uplus \overline{\overline{\mathcal{T}_2^i}}),\ \ldots,\ \Pi_2^k) \overset{t_2}{\Rightarrow} \\ \quad (\Pi_3^1,\ \ldots,\ (H_3^i,\ Hc_3^i,\ \{(\Delta_3^n,\ XS_3^n,\ (cn,\ ty)\cdot RS_3^n)\} \uplus \overline{\overline{\mathcal{T}_3^i}}),\ \ldots,\ \Pi_3^k) \\ cn\,?\,(l_2,\ p_2)\end{array}}{\begin{array}{l}(\Pi_1^1,\ \ldots,\ (H_1^i,\ Hc_1^i,\ \{(\Delta_1^n,\ XS_1^n,\ RS_1^n,\ \mathbf{receive}\,E)\} \uplus \overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi_1^k) \\ \quad \xRightarrow{t_1\,;\,t_2\,;\,cn\,?\,(l_2,\ p_2)} \\ \qquad (\Pi_3^1,\ \ldots,\ (H_3^i,\ Hc_3^i,\ \{(\Delta_3^n,\ XS_3^n,\ (l_2,\ p_2)\cdot RS_3^n)\} \uplus \overline{\overline{\mathcal{T}_3^i}}),\ \ldots,\ \Pi_3^k)\end{array}} \tag{8.31}$$

Comment: (Rule 8.31) The expression E is evaluated to provide the channel for the receive operation. The message $(l_2,\ p_2)$ is returned on the result stack.

8.2.5 Distributed Expression Evaluation

The relations for the remainder of the $\mathcal{M}\Lambda$ expressions are outlined below. These relations are essentially the same as their sequential counterparts. Therefore, as with the type allocation functions, we will define templates which can be used to perform a translation. The templates are shown in Figure 8.13.

The first template (8.32) is used to translate a relation for evaluating an expression or sub-expression. As before, the heap of the sequential relation is mapped onto the heap of the process, while the environment and stacks are mapped onto the thread. The second (8.33) and third (8.34) templates are used to translate direct heap accesses into applications of the $fetch$ function. Note that the final template returns both a value and a type on the result stack. Therefore, if the sequential abstract machine retrieves a type, followed by its corresponding value (as is usually the case), they can be combined into a single $fetch$ application.

With a little work, these templates can be used to convert all of the relations for the expressions E in Chapter 5. For brevity, the results are omitted here. As with the type functions, it may be necessary to renumber the heaps and environments in some places. A sequential evaluation trace must also be constructed where the premise of the rule contains more than one relation.

Sequential :

$$(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ E) \Rightarrow (H_2,\ \Delta_2,\ XS_2,\ (l,\ p)\cdot RS_2)$$

Distributed : (8.32)

$$(\Pi_1^1,\ \ldots,\ (H_1^i,\ Hc_1^i,\ \{(\Delta_1^n,\ XS_1^n,\ RS_1^n,\ E)\} \uplus \overline{\overline{\overline{\mathcal{T}_1^i}}}),\ \ldots,\ \Pi_1^k) \xRightarrow{t}$$
$$(\Pi_2^1,\ \ldots,\ (H_2^i,\ Hc_2^i,\ \{(\Delta_2^n,\ XS_2^n,\ (l,\ p)\cdot RS_2^n)\} \uplus \overline{\overline{\overline{\mathcal{T}_2^i}}}),\ \ldots,\ \Pi_2^k)$$

Sequential : $H_1(p) = ty$

Distributed : (8.33)

$$(\Pi_1^1,\ \ldots,\ (H_1^i,\ Hc_1^i,\ \{(\Delta_1^n,\ XS_1^n,\ RS_1^n,\ fetch(p))\} \uplus \overline{\overline{\overline{\mathcal{T}_1^i}}}),\ \ldots,\ \Pi_1^k) \xRightarrow{t}$$
$$(\Pi_2^1,\ \ldots,\ (H_2^i,\ Hc_2^i,\ \{(\Delta_2^n,\ XS_2^n,\ (ty)\cdot RS_2^n)\} \uplus \overline{\overline{\overline{\mathcal{T}_2^i}}}),\ \ldots,\ \Pi_2^k)$$

Sequential : $H_1(l) = val$

Distributed : (8.34)

$$(\Pi_1^1,\ \ldots,\ (H_1^i,\ Hc_1^i,\ \{(\Delta_1^n,\ XS_1^n,\ RS_1^n,\ fetch(l,\ p))\} \uplus \overline{\overline{\overline{\mathcal{T}_1^i}}}),\ \ldots,\ \Pi_1^k) \overset{t}{\Rightarrow}$$
$$(\Pi_2^1,\ \ldots,\ (H_2^i,\ Hc_2^i,\ \{(\Delta_2^n,\ XS_2^n,\ (val,\ ty)\cdot RS_2^n)\} \uplus \overline{\overline{\overline{\mathcal{T}_2^i}}}),\ \ldots,\ \Pi_2^k)$$

Figure 8.13: *Conversion Templates from $\mathcal{M}\Lambda$ to $\mathcal{DM}\Lambda$.*

As an example, the $\mathcal{M}\Lambda$ **app** expression is translated into distributed form in Figure 8.14. Note the sequential composition of the evaluation traces for the subexpressions: t_1 ; t_2 ; t_3 ; t_4.

Sequential :

$$\frac{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ E_1) \Rightarrow (H_2,\ \Delta_2,\ XS_2,\ (l_1,\ p_1)\cdot RS_1')\\ (H_2,\ \Delta_2,\ XS_2,\ (l_1,\ p_1)\cdot RS_1',\ E_2) \Rightarrow (H_3,\ \Delta_3,\ XS_3,\ (l_2,\ p_2)\cdot(l_1',\ p_1')\cdot RS_1'')\\ H_3(l_1') = \langle\!\langle \Delta_c,\ (x),\ E_c\rangle\!\rangle\\ (H_3,\ \Delta_c[x\ \mapsto\ (l_2,\ p_2)],\ XS_3,\ \Delta_3\cdot RS_1'',\ E_c) \Rightarrow\\ \quad (H_4,\ \Delta_4,\ XS_4,\ (l_3,\ p_3)\cdot\Delta_3'\cdot RS_1''')\end{array}}{(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ \mathbf{app}\ (E_1,\ E_2)) \Rightarrow (H_4,\ \Delta_3',\ XS_4,\ (l_3,\ p_3)\cdot RS_1''')} \tag{8.35}$$

Distributed :

$$\frac{\begin{array}{l}(\Pi_1^1,\ \ldots,\ (H_1^i,\ Hc_1^i,\ \{(\Delta_1^n,\ XS_1^n,\ RS_1^n,\ E_1)\}\uplus\overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi_1^k) \overset{t_1}{\Rightarrow}\\ \quad(\Pi_2^1,\ \ldots,\ (H_2^i,\ Hc_2^i,\ \{(\Delta_2^n,\ XS_2^n,\ (l_1,\ p_1)\cdot RS_1'^n)\}\uplus\overline{\overline{\mathcal{T}_2^i}}),\ \ldots,\ \Pi_2^k)\\ (\Pi_2^1,\ \ldots,\ (H_2^i,\ Hc_2^i,\ \{(\Delta_2^n,\ XS_2^n,\ (l_1,\ p_1)\cdot RS_1'^n,\ E_2)\}\uplus\overline{\overline{\mathcal{T}_2^i}}),\ \ldots,\ \Pi_2^k) \overset{t_2}{\Rightarrow}\\ \quad(\Pi_3^1,\ \ldots,\ (H_3^i,\ Hc_3^i,\ \{(\Delta_3^n,\ XS_3^n,\ (l_2,\ p_2)\cdot(l_1',\ p_1')\cdot RS_1''^n)\}\uplus\overline{\overline{\mathcal{T}_3^i}}),\ \ldots,\ \Pi_3^k)\\ (\Pi_3^1,\ \ldots,\ (H_3^i,\ Hc_3^i,\ \{(\Delta_3^n,\ XS_3^n,\ RS_1''^n,\ fetch(l_1',\ p_1'))\}\uplus\overline{\overline{\mathcal{T}_3^i}}),\ \ldots,\ \Pi_3^k)\\ \quad\overset{t_3}{\Rightarrow}(\Pi_4^1,\ \ldots,\ (H_4^i,\ Hc_4^i,\ \{(\Delta_4^n,\ XS_4^n,\\ \qquad(\langle\!\langle\Delta_c,\ (x),\ E_c\rangle\!\rangle,\ ty)\cdot(l_2,\ p_2)\cdot RS_1''^n)\}\uplus\overline{\overline{\mathcal{T}_4^i}}),\ \ldots,\ \Pi_4^k)\\ (\Pi_4^1,\ \ldots,\ (H_4^i,\ Hc_4^i,\ \{(\Delta_c[x\ \mapsto\ (l_2,\ p_2)],\ XS_4^n,\\ \quad\Delta_4^n\cdot RS_1''^n,\ E_c)\}\uplus\overline{\overline{\mathcal{T}_4^i}}),\ \ldots,\ \Pi_4^k)\overset{t_4}{\Rightarrow}\\ \qquad(\Pi_5^1,\ \ldots,\ (H_5^i,\ Hc_5^i,\ \{(\Delta_5^n,\ XS_5^n,\ (l_3,\ p_3)\cdot\Delta_4'^n\cdot RS_1'''^n)\}\uplus\overline{\overline{\mathcal{T}_5^i}}),\ \ldots,\ \Pi_5^k)\end{array}}{\begin{array}{l}(\Pi_1^1,\ \ldots,\ (H_1^i,\ Hc_1^i,\ \{(\Delta_1^n,\ XS_1^n,\ RS_1^n,\ \mathbf{app}\ (E_1,\ E_2))\}\uplus\overline{\overline{\mathcal{T}_1^i}}),\ \ldots,\ \Pi_1^k)\\ \quad\xRightarrow{t_1\ ;\ t_2\ ;\ t_3\ ;\ t_4}\\ \qquad(\Pi_5^1,\ \ldots,\ (H_5^i,\ Hc_5^i,\ \{(\Delta_4'^n,\ XS_5^n,\ (l_3,\ p_3)\cdot RS_1'''^n)\}\uplus\overline{\overline{\mathcal{T}_5^i}}),\ \ldots,\ \Pi_5^k)\end{array}} \tag{8.36}$$

Figure 8.14: *Function Application in $\mathcal{D}M\Lambda$.*

Earlier in this chapter (Section 8.2) it was stated that the communication traces for a correct program should always reduce to ϵ. Figure 8.15 shows that this is the case for the thread example given in Chapter 2. In this example, a thread is created on processor 2 that simply waits for an integer value on channel `c`, then returns the value, incremented by 3, on the same channel. The example concludes by sending the value 7 to the thread, and returns the response (which will be 10).

For clarity, the traces have been simplified by removing all the ϵ traces resulting from the intermediate steps in the evaluation. The first pair that is transmitted across the channel c, corresponding to the special constant 7, is denoted $(l_1,\ p_1)$. The second pair, corresponding to the special constant 10 (i.i. 7+3), is denoted $(l_2,\ p_2)$.

Concurrent ML:

```
let val c = channel()
    val _ = rfork(2,
                fn _ => send(c, receive(c) + 3))
    val _ = send(c, 7)
in
    receive(c)
end
```

$\mathcal{M}\Lambda$ Syntax:

$$
\begin{array}{ll}
(\emptyset,\ \emptyset,\ \textbf{let}\ (c,\ \text{t\_chan}(\alpha))\ =\ \textbf{con}\ \text{c\_chan} & \text{(i)} \\
\quad \textbf{in let}\ (\_,\ \text{t\_unit})\ =\ \textbf{rfork}\ (\textbf{scon}\ 2, & \\
\qquad \textbf{fn}\ (\_,\ \text{t\_unit} \to \text{t\_unit})\ = & \\
\qquad\quad \textbf{send}\ (\textbf{var}\ (c,\ \text{t\_int}), & \\
\qquad\qquad \textbf{app}\ (\text{ADD}_i,\ \textbf{tuple}\ (\textbf{receive}\ (\textbf{var}\ (c,\ \text{t\_int})),\ \textbf{scon}\ 3)))) & \text{(ii)} \\
\qquad \textbf{in let}\ (\_,\ \text{t\_unit})\ =\ \textbf{send}\ (\textbf{var}\ (c,\ \text{t\_int}),\ \textbf{scon}\ 7) & \text{(iii)} \\
\qquad\quad \textbf{in receive}\ (\text{var}\ c)) & \text{(iv)}
\end{array}
$$

Simplified Evaluation Trace:

$$
\begin{array}{ll}
\text{(i)} & \epsilon \\
\text{(ii)} & \epsilon \parallel (c\ ?\ (l_1,\ p_1)\ ;\ c\ !\ (l_2,\ p_2)) \\
\text{(iii)} & (\epsilon\ ;\ c\ !\ (l_1,\ p_1)) \parallel (c\ ?\ (l_1,\ p_1)\ ;\ c\ !\ (l_2,\ p_2)) \\
\text{(iv)} & (\epsilon\ ;\ c\ !\ (l_1,\ p_1)\ ;\ c\ ?\ (l_2,\ p_2)) \parallel (c\ ?\ (l_1,\ p_1)\ ;\ c\ !\ (l_2,\ p_2))
\end{array}
$$

Trace Reduction:

$$
\begin{array}{lll}
 & (\epsilon\ ;\ c\ !\ (l_1,\ p_1)\ ;\ c\ ?\ (l_2,\ p_2)) \parallel (c\ ?\ (l_1,\ p_1)\ ;\ c\ !\ (l_2,\ p_2)) & \\
= & (c\ !\ (l_1,\ p_1)\ ;\ c\ ?\ (l_2,\ p_2)) \parallel (c\ ?\ (l_1,\ p_1)\ ;\ c\ !\ (l_2,\ p_2)) & \text{by Rule 8.1} \\
= & c\ ?\ (l_2,\ p_2) \parallel c\ !\ (l_2,\ p_2) & \text{by Rule 8.5} \\
= & c\ !\ (l_2,\ p_2) \parallel c\ ?\ (l_2,\ p_2) & \text{by Rule 8.3} \\
= & \epsilon & \text{by Rule 8.4}
\end{array}
$$

Figure 8.15: *Trace Reduction Example.*

8.3 Implicit Concurrency

The kind of concurrency which we have defined in this chapter may be characterised as *explicit* concurrency; the programmer is required to use **fork** and **rfork** to obtain concurrent evaluation. There is another kind of concurrency that is possible in a functional setting called *implicit* or automatic concurrency. In some ways this technique is more desirable because it does not require a change to the source program. Nonetheless, implicit concurrency is entirely dependent on the opportunities for concurrency present within the program. It is perfectly possible to construct a program which has no implicit concurrency at all. Explicit concurrency allows the evaluation of the program to be tailored for the hardware on which it is to be run, while implicit concurrency does not. Implicit concurrency assumes a closely-coupled system, where the cost of creating and destroying threads, and the cost of inter-thread communication is cheap. Implicit and explicit concurrency are not mutually exclusive. It is possible to include both kinds of concurrency in a programming language. This would be appropriate for a network of multiprocessor machines. In this section we briefly outline a technique for implicit concurrency in the $\mathcal{M}\Lambda$ language.

Implicit concurrency is particularly suited to lazy functional languages, e.g. Haskell [JH93] and UFO [SKA94]. However, it can also be applied to strict functional languages such as Standard ML. Our approach to implicit concurrency is similar to that of Parallel SML [Ham91], though our semantic model is entirely different. By definition, a function is considered *strict* in an argument if the result of the function is $\perp$ (non-termination) when only that argument is $\perp$. It follows that every strict argument must be evaluated in order to obtain the result of the function. For example, an arithmetic addition function is strict in both its arguments, since both are necessary to compute the result, and the result is $\perp$ if either argument is $\perp$. It is therefore possible to evaluate the arguments in parallel without introducing non-termination. In Standard ML, every function is strict in its arguments. Thus, we can introduce implicit concurrency into $\mathcal{M}\Lambda$ by way of function application.

Unfortunately there is a significant caveat to implicit concurrency. We can only evaluate function arguments in parallel if they do not involve any side-effects, i.e. referential transparency is preserved. This means that we must make two simplifications to the $\mathcal{M}\Lambda$ language, namely the removal of references and exceptions. It may appear that we do not need to remove side-effects from the language if their evaluation can be isolated. We may then enforce sequential evaluation on these areas. Unfortunately, these areas cannot be identified purely by their definition. Side-effects can readily 'pollute' the purely functional parts of the program, for example, applying a pure higher-order function (e.g. `map`) to a function with side-effects (e.g. `inc`). Determining all the side-effecting areas of a program should be possible using static analysis techniques, e.g. a type system with annotations such as in [Kı99] which presents a technique for detecting mobile functions. However, this is beyond the scope of the thesis. For brevity, we have chosen to eliminate references from the language for the purpose of illustrating implicit concurrency.

The problem with exceptions occurs when two expressions which are evaluated in parallel can raise an exception. The result is entirely dependent on which expression terminates first. Clearly, this will destroy referential transparency and introduce an undesirable element of non-determinism into the language. Exceptions are a very powerful technique in ML, and their removal is undesirable. Fortunately, the removal of exceptions from the language is not entirely necessary. It is possible to retain exceptions in the language, if we change the semantics of exception handling. The new semantics are essentially the same as those required for an implementation of exceptions in a lazy functional language. The first change is to treat exceptions as *values*, as opposed to a change in control flow. The second change is to define a *set* of exceptions which can be raised, rather than a single exception. The final stage is to determine which exception in the set has actually been raised. There are a number of possibilities which may be used here: we can make a non-deterministic choice between the exceptions [MLP99], alternatively we can impose a prioritisation on the exceptions [Ham91], or we can simply handle all of the exceptions as the same. For brevity, we have eliminated exceptions from $\mathcal{M}\Lambda$ as we merely wish to outline implicit concurrency in $\mathcal{M}\Lambda$. However, in an actual implementation, it would be desirable to implement one of the schemes above.

We now sketch an abstract machine model which can perform implicitly concurrent evaluation of $\mathcal{M}\Lambda$ expressions. We do not include the explicit concurrency (threading) operations in our model. The static semantics of the language remain the same as in Chapter 3, with the exclusion of references, exceptions, and threads. The abstract machine syntax is illustrated in Figure 8.16. The definition of the heap H, environment Δ and result stack RS are identical to the sequential abstract machine and have been omitted.

The state of the abstract machine is defined by a single heap H and a multi-set of micro-threads $\overline{\overline{\mu}}$. The micro-threads are more lightweight than the threads $\mathcal{T}$ defined earlier in this chapter. In a model with both explicit and implicit parallelism, each thread $\mathcal{T}$ would consist of a set of micro-threads μ. A separate micro-thread is used for the evaluation of every expression and sub-expression, in a program. There is no direct communication (i.e. via channels) between micro-threads, rather the result of each micro-thread is available to the thread which created it.

Machine State	M	$::=$	$(H,\ \overline{\overline{\mu}})$
Micro Thread	μ	$::=$	$(\Delta,\ RS)$
Trace	t	$::=$	$\epsilon \mid t_1\ ;\ t_2 \mid t_1 \parallel t_2$

Figure 8.16: *Implicitly Concurrent Abstract Machine.*

The abstract machine is defined by relations of the form: $(H_1,\ (\Delta_1,\ RS_1,\ E) \uplus \overline{\overline{\mu_1}}) \overset{t}{\Rightarrow} (H_2,\ (\Delta_2,\ (l,\ p) \cdot RS_2) \uplus \overline{\overline{\mu_2}})$. As in the distributed abstract machine, each micro-thread has access to a shared heap, and a trace t is maintained to indicate the evaluation order of the micro-threads. The concurrent evaluation of threads is defined in Figure 8.17.

$$\frac{\mu_1 \overset{t_1}{\Longrightarrow} \mu_3 \qquad \overline{\overline{\mu_2}} \overset{t_2}{\Longrightarrow} \overline{\overline{\mu_4}}}{\{\mu_1\} \uplus \overline{\overline{\mu_2}} \overset{t_1 \parallel t_2}{\Longrightarrow} \{\mu_3\} \uplus \overline{\overline{\mu_4}}} \tag{8.37}$$

Figure 8.17: *Concurrent Evaluation of Micro Threads.*

For brevity we simply define relations for the single and multiple argument **app** expressions, which are responsible for introducing implicit concurrency. The most significant part of each rule is the trace at the conclusion of the rule. The trace for a function of a single argument (Rule 8.38) states that the evaluation of the function expression E_1 with trace t_1 may occur in parallel with the evaluation of the argument expression E_2. However, the evaluation of these expressions must be complete before the function E_c with trace t_3 is evaluated. This ordering is enforced by the sequential composition.

$$\frac{\begin{array}{l}(H_1,\ (\Delta_1,\ RS_1,\ E_1) \uplus \overline{\overline{\mu_1}}) \overset{t_1}{\Rightarrow} (H_2,\ (\Delta_2,\ (l_1,\ p_1) \cdot RS_1) \uplus \overline{\overline{\mu_2}}) \\ (H_2,\ (\Delta_2,\ (l_1,\ p_1) \cdot RS_1,\ E_2) \uplus \overline{\overline{\mu_2}}) \overset{t_2}{\Rightarrow} \\ \quad (H_3,\ (\Delta_3,\ (l_2,\ p_2) \cdot (l_1,\ p_1) \cdot RS_1) \uplus \overline{\overline{\mu_3}}) \\ H_3(l_1') = \langle\!\langle \Delta_c,\ (x),\ E_c \rangle\!\rangle \\ (H_3,\ (\Delta_c[x \mapsto (l_2,\ p_2)],\ \Delta_3 \cdot RS_1,\ E_c) \uplus \overline{\overline{\mu_3}}) \overset{t_3}{\Rightarrow} \\ \quad (H_4,\ (\Delta_4,\ (l_3,\ p_3) \cdot \Delta_3 \cdot RS_1) \uplus \overline{\overline{\mu_4}})\end{array}}{\begin{array}{l}(H_1,\ (\Delta_1,\ RS_1,\ \mathbf{app}\ (E_1,\ E_2)) \uplus \overline{\overline{\mu_1}}) \overset{(t_1 \parallel t_2)\ ;\ t_3}{\Longrightarrow} \\ \quad (H_4,\ (\Delta_3,\ XS_4,\ (l_3,\ p_3) \cdot RS_1) \uplus \overline{\overline{\mu_4}})\end{array}} \tag{8.38}$$

$$
\frac{
\begin{array}{l}
(H_1,\ (\Delta_1,\ RS_1,\ E_1) \uplus \overline{\overline{\mu_1}}) \overset{t_1}{\Rightarrow} (H_2,\ (\Delta_2,\ (l_1,\ p_1) \cdot RS_1) \uplus \overline{\overline{\mu_2}}) \\
(H_2,\ (\Delta_2,\ (l_1,\ p_1) \cdot RS_1,\ E_2^1) \uplus \overline{\overline{\mu_2}}) \overset{t_2^1}{\Rightarrow} (H_3,\ (\Delta_3,\ RS_2) \uplus \overline{\overline{\mu_3}}) \ \cdots \\
\quad (H_{k+1},\ (\Delta_{k+1},\ RS_k,\ E_2^k) \uplus \overline{\overline{\mu_{k+1}}}) \overset{t_2^k}{\Rightarrow} \\
\qquad (H_{k+2},\ (\Delta_{k+2},\ (l_2^k,\ p_2^k) \cdot (l_2^1,\ p_2^1) \cdot (l_1,\ p_1) \cdot RS_1) \uplus \overline{\overline{\mu_{k+2}}}) \\
H_{k+2}(l_1) = \langle\!\langle \Delta_c,\ \overline{x}^k,\ E_c \rangle\!\rangle \\
(H_{k+2},\ (\Delta_c[x^1 \ \mapsto\ (l_2^1,\ p_2^1),\ \ldots,\ x^k \ \mapsto\ (l_2^k,\ p_2^k)],\ \Delta_{k+2} \cdot RS_1,\ E_c) \uplus \overline{\overline{\mu_{k+2}}}) \overset{t_3}{\Rightarrow} \\
\quad (H_{k+3},\ (\Delta_{k+3},\ (l_3,\ p_3) \cdot \Delta'_{k+2} \cdot RS_1) \uplus \overline{\overline{\mu_{k+3}}})
\end{array}
}{
\begin{array}{l}
(H_1,\ (\Delta_1,\ RS_1,\ \mathbf{app}\ (E_1,\ \overline{E_2}^k)) \uplus \overline{\overline{\mu_1}}) \xrightarrow{(t_1 \| t_2^1 \| \cdots \| t_2^k)\ ;\ t_3} \\
\quad (H_{k+3},\ (\Delta_{k+2},\ RS_1) \uplus \overline{\overline{\mu_{k+3}}})
\end{array}
}
\tag{8.39}
$$

The remaining $\mathcal{M}\Lambda$ expressions are simply evaluated in a sequential manner as before. We have yet to consider the relationship between implicit parallelism and code-replacement. However, there are unlikely to be many conflicts as code-replacement is primarily a heap transformation, and the definition of the heap remains unchanged.

Chapter 9

Distributed Code-Replacement

In this chapter we complete our definition of distributed code-replacement. The chapter is divided into two parts: in the first part we define a distributed garbage collection algorithm, and in the second part we extend this algorithm with code-replacement. Distributed garbage collection and distributed code-replacement may be considered as generalisations of the sequential definitions presented earlier. In general, distributed garbage collection algorithms are notoriously difficult to define correctly. Thus, in Chapter 10 we perform a mechanical verification of our definition.

In the previous chapter we described the LEMMA interface [MS95b] as a Distributed Shared Memory (DSM) platform with distributed garbage collection. Therefore, this chapter may be considered as a completion of the LEMMA definition. Recall that the LEMMA interface is simply a function-level specification and does not impose any constraints on the implementation. Consequently, the choice of distributed garbage collection algorithm is left open.

The provision of efficient algorithms for distributed garbage collection has been, and continues to be, a very active area of research. There are many differing techniques, ranging from the very simple, e.g. where one process garbage collects while another executes, up to complex multi-generational schemes. A good survey of distributed garbage collection techniques is presented in [AR98].

We have chosen to use an extension of the copying collection algorithm of Chapter 6. The technique presented here is not the most efficient distributed algorithm. Nonetheless, it appears to be an appropriate choice as it extends naturally from the sequential case, and has a number of desirable properties for code-replacement, e.g. rollback. Furthermore, we have retained the tag-free style which may potentially increase the efficiency of the algorithm. A number of other improvements to the basic algorithm are discussed at the end of the chapter.

9.1 Distributed Garbage Collection

We begin by presenting an informal description of the distributed garbage collection algorithm. We then proceed to define the algorithm formally by exhibiting an abstract machine with the same behaviour. The high-level view of the algorithm is illustrated in Figure 9.1.

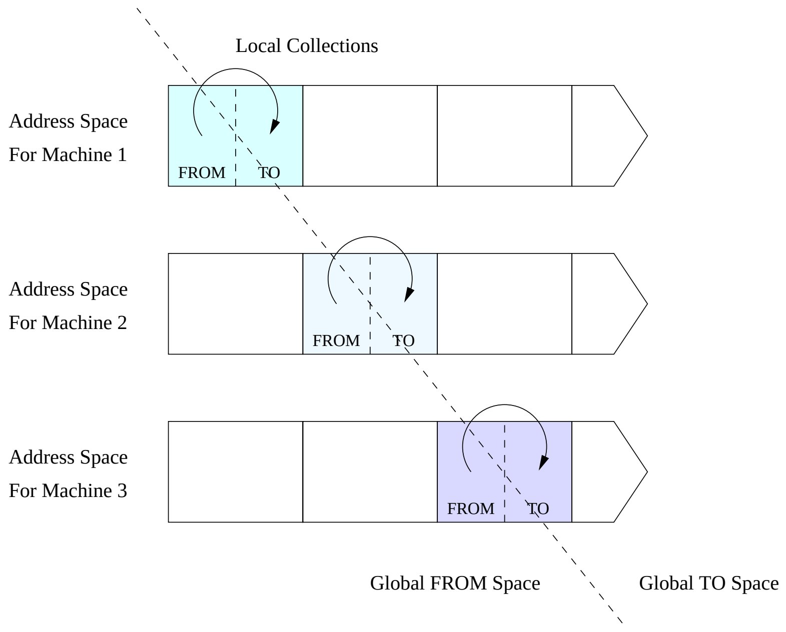

Local Collections
Address Space For Machine 1
FROM
TO
Address Space For Machine 2
FROM
TO
Address Space For Machine 3
FROM
TO
Global FROM Space
Global TO Space

Figure 9.1: *Distributed Garbage Collection.*

The collection is performed over the distributed shared memory, which we will call the global address space. Each semi-space of the global address space managed by each machine is divided in two to constitute local *from* and *to* spaces. The global *from* and *to* spaces for the collection are considered to be the concatenation of these local spaces. However, each machine only garbage collects in its own local space.

Garbage collection begins with a global synchronisation; all of the machines perform garbage collection in parallel, but asynchronously. The task of each machine is to ensure that all of the objects in the global *from* space that are reachable from its own roots have been copied into the global *to* space. Once all of the machines have finished, the garbage collection is complete and each machine resumes executing in its local *to* space. The garbage collection algorithm executed by each machine is the same as the sequential variant with the following additions:

1. When machine A encounters a pointer to an object managed by machine B, it sends a message containing the pointer to machine B.

2. If the object has already been copied, machine B returns the updated pointer, so machine A can update the object that it was scanning.

3. If the object has not been copied, machine B proceeds with the copy and returns the updated pointer to machine A. During the copy, machine B may encounter further remote pointers, in which case the above steps are repeated.

The protocol has the property that an object is always copied by the machine that first created it, even if that machine no longer has a reference to it. It must also be noted that the garbage collection invalidates all of the cached copies of objects on other machines.

9.1.1 Abstract Machine Definition

We will now present an abstract machine definition which describes the Distributed Garbage Collection (DGC) algorithm. The syntax of this abstract machine is shown in Figure 9.2.

Machine State	DGC	$::=$	π^k
DGC Process	π	$::=$	$(Hf,\ Ht,\ GT,\ \overline{\overline{T}})$
DGC Tables	GT	$::=$	$(RM,\ PF,\ LF,\ TE)$
DGC Threads	T	$::=$	$V \mid K \mid L \mid P$
		$\mid$	$W_K(S_1,\ S_2) \mid W_L(l_1,\ p_1,\ l_2,\ p_2) \mid W_P(p_1,\ p_2)$
Trace	t	$::=$	$\epsilon \mid t_1 \ ;\ t_2 \mid t_1 \parallel t_2 \mid cn\ !\ m \mid cn\ ?\ m$
Message	m	$::=$	$(p) \mid (l,\ p) \mid (\Delta) \mid (RM)$

Figure 9.2: *Abstract Machine Syntax.*

The organisation of our abstract machine for garbage collection is very similar to the abstract machine for distributed evaluation defined in Chapter 8. This similarity is intentional as there is a one-to-one correspondence between evaluation processes Π and garbage collection processes π. In an actual implementation the two machines would be combined. The combined machine would alternate between the evaluation and garbage collection processes during cycles of program execution and garbage collection. Figure 9.3 defines the parallel execution of garbage collection processes and threads.

Each garbage collection process π is a separate copy of the sequential algorithm. Therefore, each process contains a garbage collection state as defined

$$\frac{\pi_1^1 \xrightarrow{t_1}_{gc} \pi_2^1 \quad \cdots \quad \pi_1^k \xrightarrow{t_k}_{gc} \pi_2^k}{(\pi_1^1, \ldots, \pi_1^k) \xrightarrow{t_1 \| \cdots \| t_k}_{gc} (\pi_2^1, \ldots, \pi_2^k)} \tag{9.1}$$

$$\frac{T_1 \xrightarrow{t_1}_{gc} T_3 \qquad \overline{\overline{T_2}} \xrightarrow{t_2}_{gc} \overline{\overline{T_4}}}{\{T_1\} \uplus \overline{\overline{T_2}} \xrightarrow{t_1 \| t_2}_{gc} \{T_3\} \uplus \overline{\overline{T_4}}} \tag{9.2}$$

Figure 9.3: *Execution of $\mathcal{D}GC$ Processes and Threads.*

in Chapter 6 (Section 6.2), i.e. each process has a *from* heap Hf^i, a *to* heap Ht^i, and its own copy of the garbage collection tables GT^i. In order to implement the sequential algorithm in a multi-threaded manner, there are separate threads for garbage collecting environments V, stacks K, values L, and types P respectively. Messages can consist of pointers, locations, and environments as defined in Chapter 5, and replacement maps as defined in Chapter 7.

9.1.2 Collecting Types and Values

The threads for collecting values and types are named L^i and P^i, where $0 < i \leq k$, i.e. each garbage collection process π has a single L and P thread associated with it. Each garbage collection process is only responsible for the collection of data that is contained within its own heap. For data that is contained within the heap of another process, it is necessary to send the pointer across a channel to this process for collection. Hence, each thread L^i, and P^i and has a pair of channels associated with it; one for sending pointers to the thread for collection, and one along which the reply, containing the updated pointer, is received. In Chapter 10 we show why a single channel for both sending and receiving will not work. We use underlining to distinguish between sending and receiving channels. The channels for sending pointers to the L^i and P^i threads are named cn_L^i and cn_P^i, and the channels for receiving pointers from these threads are named $\underline{cn}_L^i$ and $\underline{cn}_P^i$. When an external pointer is encountered during garbage collection, it is simply sent along a channel to the appropriate thread which returns an updated pointer along the other channel. The thread is determined using the *num* operation defined in Chapter 8. For example, a type referenced by the pointer p_1 is garbage collected by sending it to the correct L thread $cn_L^i \ ! \ (p_1)$, where $i = num(p_1)$. The collected pointer p_2 is subsequently retrieved from the same thread $\underline{cn}_L^i \ ? \ (p_2)$. For convenience, we will use the shorthand $L^i \ ! \ (p_1) \ ? \ (p_2)$ to denote a send, followed by a receive to thread L^i. In order to simplify the garbage collection rules, a distinction is not made between local and remote data. All type-pointers and value-locations are sent along channels, even if the data is contained within the heap of the same process, i.e. when i is the local process. In an actual implementation it would be desirable to optimise the local case.

It may be the case that two (or more) threads attempt to communicate with another thread at the same time. In this case, a non-deterministic choice is made between the communicating threads. Only one thread is permitted to communicate along a channel at once. The remaining threads are blocked until the communication is finished. Communication with a thread in our collector is always done by performing a send operation followed by a receive. By maintaining this strict ordering, we ensure that the correct thread always receives the reply.

A copying operation may require the cooperation of a number of threads. This could easily lead to a deadlock owing to the blocking nature of the communication, e.g. two threads may become blocked waiting for each other to finish. One solution would be to fork a new thread every time. However, each of these threads would require a separate pair of communication channels, and this would considerably complicate the collection algorithm. The solution we adopt involves the use of two different kinds of threads, namely *server* and *worker* threads.

The threads L^i and P^i which we discussed previously are the server threads for the values and types respectively. These threads are primarily responsible for maintaining the communication channels. When a pointer is received on a communication channel, the server simply reserves and returns another pointer on the other channel while a separate worker thread is forked to perform the actual copying of the data. Server threads do not communicate directly with each other, a request to a server is always made from a worker thread. By removing blocking from the server threads in this manner, we avoid the possibility of deadlocks, as we will demonstrate in Chapter 10.

The worker threads are denoted W_L and W_P. Unlike the server threads, which are static until the end of the collection, worker threads are created and destroyed dynamically throughout. A worker thread is passed a list of arguments when it is created. The worker thread for copying a type is created as follows: $W_P(p_1,\ p_2)$. This thread performs the collection of the type referenced by p_1 in the local *from* heap into the *to* heap referenced by p_2. Similarly, the worker thread for copying a value is created as $W_L(l_1,\ p_1,\ l_2,\ p_2)$. This thread performs the collection of the value referenced by $(l_1,\ p_1)$ into the location referenced by $(l_2,\ p_2)$. Worker threads may block when communicating with server threads. We will prove in Chapter 10 that all the worker threads will eventually terminate.

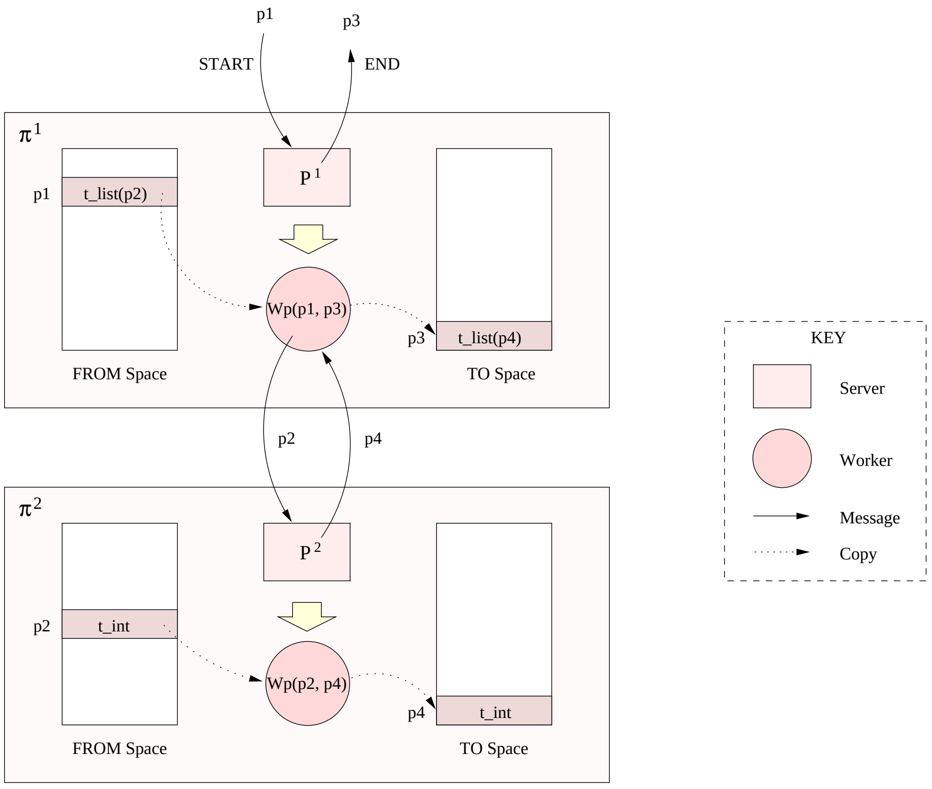


Figure 9.4: *Distributed Garbage Collection Example.*

An example distributed garbage collection with two processes is illustrated in Figure 9.4. In the example, an integer-list type is garbage collected. Initially, the list type t_list is referenced by the type-pointer p_1 and is contained within the *from* heap of process π^1. The list type contains a pointer p_2 to the integer type t_int contained within the *from* heap of process π^2. This type will be copied into the *to* heap on π^1 and π^2. The figure should be read in sequence from top to bottom. Garbage collection is initiated by sending the pointer p_1 to the server thread P^1 on process π^1. The server thread reserves and returns pointer p_3 on the *to* heap of process π^1. A worker thread $W_P(p_1,\ p_3)$ is created to perform the copy of the list type. During the copy operation, the worker thread encounters the pointer p_2 to the integer type. This pointer is sent to the server thread P^2 on process π^2. The server thread subsequently reserves a pointer p_4 on the *to* heap of process π^2 and returns this to the worker thread. Now a new worker thread $W_P(p_2,\ p_4)$ is created on process π^2 to perform the copy of the integer type. Meanwhile, the worker thread on process π^1 completes the copy of the list type with the updated pointer p_4. Once the worker thread on process π^2 has completed the copy of the integer type, the garbage collection is complete. The type-pointer p_3 references the garbage collected type in the *to* heap of process π^1.

We will now formally define the server and worker threads for types and values. As in Chapter 8, this is achieved by presenting a series of relations which define the behaviour of a single thread on process i, where $0 < i \leq k$. A computational trace t is also constructed for recording any external interactions.

The server thread for types P^i is defined in Rules 9.3 and 9.4. The thread will wait until it receives a type pointer $cn^i_P \ ?\ (p_1)$. In Rule 9.3, the type referenced by p_1 has already been collected. The corresponding entry in the forwarding table is therefore returned $\underline{cn^i_P} \ !\ (PF^i(p_1))$. In Rule 9.4 the type has not been collected. A pointer p_2 is reserved in Ht, to hold the collected type, and returned $\underline{cn^i_P} \ !\ (p_2)$. At the same time, a worker thread $W_P(p_1,\ p_2)$ is created to copy the type, and a mapping $p_1 \ \mapsto\ p_2$ is added to the forwarding table to prevent the type being collected again. Note that the server is still present in the set of threads at the end of the rule. This has the effect of restarting the type server.

$$\frac{\begin{array}{l} GT^i = (RM^i,\ PF^i,\ LF^i,\ TE^i) \\ t_1 = cn^i_P \ ?\ (p_1) \qquad p_1 \in \text{Dom}\ PF^i \qquad t_2 = \underline{cn^i_P} \ !\ (PF^i(p_1)) \end{array}}{\begin{array}{l} (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ P^i \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \overset{t_1\ ;\ t_2}{\Longrightarrow}_{gc} \\ \quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ P^i \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \end{array}} \tag{9.3}$$

$$\frac{\begin{array}{l} GT^i_1 = (RM^i,\ PF^i,\ LF^i,\ TE^i) \\ t_1 = cn^i_P \ ?\ (p_1) \qquad p_1 \notin \text{Dom}\ PF^i \qquad Ht^i \uparrow p_2 \qquad t_2 = \underline{cn^i_P} \ !\ (p_2) \\ GT^i_2 = (RM^i,\ PF^i[p_1 \ \mapsto\ p_2],\ LF^i,\ TE^i) \end{array}}{\begin{array}{l} (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i_1,\ P^i \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \overset{t_1\ ;\ t_2}{\Longrightarrow}_{gc} \\ \quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i_2,\ P^i \uplus W_P(p_1,\ p_2) \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \end{array}} \tag{9.4}$$

The value server L^i is defined in Rules 9.5 and 9.6. The value server is essentially the same as the type server. In an actual implementation it may be prudent to combine these into a single server which acts according to the type of message received. However, separating the servers here makes the definition clearer.

$$\frac{\begin{array}{l} GT^i = (RM^i,\ PF^i,\ LF^i,\ TE^i) \\ t_1 = cn^i_L \ ?\ (l_1,\ p_1) \qquad l_1 \in \text{Dom}\ LF^i \qquad t_2 = \underline{cn^i_L} \ !\ (LF^i(l_1),\ PF^i(p_1)) \end{array}}{\begin{array}{l} (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ L^i \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \overset{t_1\ ;\ t_2}{\Longrightarrow}_{gc} \\ \quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ L^i \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \end{array}} \tag{9.5}$$

$$\frac{\begin{array}{l} GT^i_1 = (RM^i,\ PF^i,\ LF^i,\ TE^i) \\ t_1 = cn^i_L \ ?\ (l_1,\ p_1) \qquad l_1 \notin \text{Dom}\ LF^i \qquad Ht^i \uparrow l_2 \qquad t_2 = \underline{cn^i_L} \ !\ (l_2,\ PF^i(p_1)) \\ GT^i_2 = (RM^i,\ PF^i,\ LF^i[l_1 \ \mapsto\ l_2],\ TE^i) \end{array}}{\begin{array}{l} (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i_1,\ L^i \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \overset{t_1\ ;\ t_2}{\Longrightarrow}_{gc} \\ \quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i_2,\ L^i \uplus W_L(l_1,\ p_1,\ l_2,\ PF^i(p_1)) \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \end{array}} \tag{9.6}$$

The worker thread for types $W_P(p_1,\ p_2)$ is defined below. There are separate rules for each of the types in the $\mathcal{M}\Lambda$ language. Any type pointers that are encountered inside a type are collected by sending them to the appropriate type server P. The heap is accessed directly, instead of going via the $fetch$ operation as was done in Chapter 8. This is possible because the types will always be contained within the local heap.

The rules defined below are essentially identical to those presented in Chapter 6; where the sequential algorithm performs the collection of a type pointer $(Hf,\ Ht_1,\ PF_1,\ LF,\ p_1) \Rightarrow_{\mathrm{gc}} (Hf,\ Ht_2,\ PF_2,\ LF,\ p_2)$, the distributed algorithm communicates with a server $P^i\ !\ (p_1)\ ?\ (p_2)$. Note that if there is more than one type pointer, e.g. Rule 9.9, we collect them in parallel. This allows us to overcome the sequential bottleneck discussed at the end of Chapter 6.

$$\frac{Hf^i(p_1) = tn}{\begin{array}{l}(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ W_P(p_1,\ p_2) \uplus \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k) \overset{\epsilon}{\Rightarrow}_{gc}\\ \quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i[p_2\ \mapsto\ tn],\ GT^i,\ \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k)\end{array}} \tag{9.7}$$

$$\frac{Hf^i(p_1) = tn(p_3) \qquad t = P^{num(p_3)}\ !\ (p_3)\ ?\ (p_4)}{\begin{array}{l}(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ W_P(p_1,\ p_2) \uplus \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k) \overset{t}{\Longrightarrow}_{gc}\\ \quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i[p_2\ \mapsto\ tn(p_4)],\ GT^i,\ \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k)\end{array}} \tag{9.8}$$

$$\frac{Hf^i(p_1) = \overline{p_3}^{\,k} \qquad t = P^{num(p_3^1)}\ !\ (p_3^1)\ ?\ (p_4^1) \parallel \cdots \parallel P^{num(p_3^k)}\ !\ (p_3^k)\ ?\ (p_4^k)}{\begin{array}{l}(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ W_P(p_1,\ p_2) \uplus \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k) \overset{t}{\Longrightarrow}_{gc}\\ \quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i[p_2\ \mapsto\ \overline{p_4}^{\,k}],\ GT^i,\ \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k)\end{array}} \tag{9.9}$$

$$\frac{Hf^i(p_1) = p_3 \to p_4 \qquad t = P^{num(p_3)}\ !\ (p_3)\ ?\ (p_5) \parallel P^{num(p_4)}\ !\ (p_4)\ ?\ (p_6)}{\begin{array}{l}(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ W_P(p_1,\ p_2) \uplus \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k) \overset{t}{\Longrightarrow}_{gc}\\ \quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i[p_2\ \mapsto\ p_5 \to p_6],\ GT^i,\ \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k)\end{array}} \tag{9.10}$$

$$\frac{Hf^i(p_1) = \alpha}{\begin{array}{l}(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ W_P(p_1,\ p_2) \uplus \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k) \overset{\epsilon}{\Rightarrow}_{gc}\\ \quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i[p_2\ \mapsto\ \alpha],\ GT^i,\ \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k)\end{array}} \tag{9.11}$$

$$\frac{\begin{array}{l}Hf^i(p_1) = \langle\!\langle TE_1,\ p_3\rangle\!\rangle \qquad TE_1 = \{\alpha^1\ \mapsto\ p_4^1,\ \ldots,\ \alpha^k\ \mapsto\ p_4^k\}\\ t_1 = P^{num(p_4^1)}\ !\ (p_4^1)\ ?\ (p_5^1) \parallel \cdots \parallel P^{num(p_4^k)}\ !\ (p_4^k)\ ?\ (p_5^k)\\ TE_2 = \{\alpha^1\ \mapsto\ p_5^1,\ \ldots,\ \alpha^k\ \mapsto\ p_5^k\} \qquad t_2 = P^{num(p_3)}\ !\ (p_3)\ ?\ (p_6)\end{array}}{\begin{array}{l}(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ W_P(p_1,\ p_2) \uplus \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k) \overset{t_1 \parallel t_2}{\Longrightarrow}_{gc}\\ \quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i[p_1\ \mapsto\ \langle\!\langle TE_2,\ p_6\rangle\!\rangle],\ GT^i,\ \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k)\end{array}} \tag{9.12}$$

We define the value worker thread $W_L(l_1,\ p_1,\ l_2,\ p_2)$ below. There are separate rules for each of the values in the language. The type information referenced by p_1 is used to guide the collection, i.e. a tag-free collection. Any locations that are encountered during collection are sent to the appropriate value server L as before.

$$\frac{Hf^i(p_1) = tn}{\begin{array}{l}(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ W_L(l_1,\ p_1,\ l_2,\ p_2) \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \overset{\epsilon}{\Rightarrow}_{gc} \\ \quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i[l_2 \ \mapsto\ Hf^i(l_1)],\ GT^i,\ \overline{\overline{T^i}}),\ \ldots,\ \pi^k)\end{array}} \tag{9.13}$$

$$\frac{Hf^i(p_1) = tn(p_2) \qquad Hf^i(l_1) = con(l_3) \qquad t = L^{num(l_3)}\ !\ (l_3,\ p_2)\ ?\ (l_4,\ p_2)}{\begin{array}{l}(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ W_L(l_1,\ p_1,\ l_2,\ p_2) \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \overset{t}{\Rightarrow}_{gc} \\ \quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i[l_2 \ \mapsto\ con(l_4)],\ GT^i,\ \overline{\overline{T^i}}),\ \ldots,\ \pi^k)\end{array}} \tag{9.14}$$

$$\frac{\begin{array}{l}Hf^i(p_1) = \overline{p_2}^{\,k} \qquad Hf^i(l_1) = \overline{l_3}^{\,k} \\ t = L^{num(l_3^1)}\ !\ (l_3^1,\ p_2^1)\ ?\ (l_4^1,\ p_2^1) \parallel \cdots \parallel L^{num(l_3^k)}\ !\ (l_3^k,\ p_2^k)\ ?\ (l_4^k,\ p_2^k)\end{array}}{\begin{array}{l}(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ W_L(l_1,\ p_1,\ l_2,\ p_2) \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \overset{t}{\Rightarrow}_{gc} \\ \quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i[l_2 \ \mapsto\ (l_4^1,\ \ldots,\ l_4^k)],\ GT^i,\ \overline{\overline{T^i}}),\ \ldots,\ \pi^k)\end{array}} \tag{9.15}$$

$$\frac{Hf^i(p_1) = p_2 \rightarrow p_3 \qquad Hf^i(l_1) = \langle\!\langle \Delta_1,\ \overline{x}^{\,k},\ E \rangle\!\rangle \qquad t = V^i\ !\ \Delta_1\ ?\ \Delta_2}{\begin{array}{l}(\pi_1^1,\ \ldots,\ (Hf_1^i,\ Ht_1^i,\ GT_1^i,\ W_L(l_1,\ p_1,\ l_2,\ p_2) \uplus \overline{\overline{T_1^i}}),\ \ldots,\ \pi_1^k) \overset{t}{\Rightarrow}_{gc} \\ \quad (\pi_2^1,\ \ldots,\ (Hf_2^i,\ Ht_2^i[l_2 \ \mapsto\ \langle\!\langle \Delta_2,\ \overline{x}^{\,k},\ E \rangle\!\rangle],\ GT_2^i,\ \overline{\overline{T_2^i}}),\ \ldots,\ \pi_2^k)\end{array}} \tag{9.16}$$

$$\frac{Hf^i(p_1) = p_2 \rightarrow p_3 \qquad Hf^i(l_1) = \Omega}{\begin{array}{l}(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ W_L(l_1,\ p_1,\ l_2,\ p_2) \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \overset{\epsilon}{\Rightarrow}_{gc} \\ \quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i[l_2 \ \mapsto\ \Omega],\ GT^i,\ \overline{\overline{T^i}}),\ \ldots,\ \pi^k)\end{array}} \tag{9.17}$$

$$\frac{\begin{array}{l}Hf^i(p_1) = \alpha \qquad GT_1^i = (RM^i,\ PF^i,\ LF^i,\ TE^i) \\ (\pi_1^1,\ \ldots,\ (Hf_1^i,\ Ht_1^i,\ GT_1^i,\ W_L(l_1,\ TE^i(\alpha),\ l_2,\ p_2) \uplus \overline{\overline{T_1^i}}),\ \ldots,\ \pi_1^k) \overset{t}{\Rightarrow}_{gc} \\ \quad (\pi_2^1,\ \ldots,\ (Hf_2^i,\ Ht_2^i,\ GT_2^i,\ \overline{\overline{T_2^i}}),\ \ldots,\ \pi_2^k)\end{array}}{\begin{array}{l}(\pi_1^1,\ \ldots,\ (Hf_1^i,\ Ht_1^i,\ GT_1^i,\ W_L(l_1,\ p_1,\ l_2,\ p_2) \uplus \overline{\overline{T_1^i}}),\ \ldots,\ \pi_1^k) \overset{t}{\Rightarrow}_{gc} \\ \quad (\pi_2^1,\ \ldots,\ (Hf_2^i,\ Ht_2^i,\ GT_2^i,\ \overline{\overline{T_2^i}}),\ \ldots,\ \pi_2^k)\end{array}} \tag{9.18}$$

$$\frac{\begin{array}{l}Hf^i(p_1) = \langle\!\langle TE_3,\ p_3 \rangle\!\rangle \qquad GT_1^i = (RM_1^i,\ PF_1^i,\ LF_1^i,\ TE_3 \cup TE_1^i) \\ (\pi_1^1,\ \ldots,\ (Hf_1^i,\ Ht_1^i,\ GT_1^i,\ W_L(l_1,\ p_3,\ l_2,\ p_2) \uplus \overline{\overline{T_1^i}}),\ \ldots,\ \pi_1^k) \overset{t}{\Rightarrow}_{gc} \\ \quad (\pi_2^1,\ \ldots,\ (Hf_2^i,\ Ht_2^i,\ (RM_2^i,\ PF_2^i,\ LF_2^i,\ TE_2^i),\ \overline{\overline{T_2^i}}),\ \ldots,\ \pi_2^k)\end{array}}{\begin{array}{l}(\pi_1^1,\ \ldots,\ (Hf_1^i,\ Ht_1^i,\ (RM_1^i,\ PF_1^i,\ LF_1^i,\ TE_1^i), \\ \quad W_L(l_1,\ p_1,\ l_2,\ p_2) \uplus \overline{\overline{T_1^i}}),\ \ldots,\ \pi_1^k) \overset{t}{\Rightarrow}_{gc} \\ \qquad (\pi_2^1,\ \ldots,\ (Hf_2^i,\ Ht_2^i,\ (RM_2^i,\ PF_2^i,\ LF_2^i,\ TE_2^i - TE_3),\ \overline{\overline{T_2^i}}),\ \ldots,\ \pi_2^k)\end{array}} \tag{9.19}$$

9.1.3 Collecting Environments and Stacks

The threads for collecting types and values perform the majority of the work of distributed garbage collection. However, before these threads can be used we need a means of decomposing the roots, that is the environments and stacks, and a means of reassembling them again afterwards. To achieve this, we introduce environment V and stack K threads. As before, each garbage collection process π will have a single associated V^i and K^i thread. We also define pairs of channels cn^i_V, $\underline{cn}^i_V$ and cn^i_K, $\underline{cn}^i_K$ for communicating with these threads. The difference from L^i and P^i is that communication will always take place locally. This is because environments and stacks are not heap-allocated. Hence, it makes sense to always collect them within the local process.

The environment thread V^i is a server thread. However, it does not require a corresponding worker thread. At first it may appear that this is a mistake, since the environment will reference closures that will themselves contain environments. Thus, it would appear that a single environment thread is insufficient and will cause a deadlock when a closure is encountered. However, with some thought, once can become convinced that this will not be a problem. The environment will contain a reference $(l,\ p)$ to a closure, not the actual closure itself. We will therefore involve the L^i and P^i server threads in the collection. These server threads will return an immediate result and fork off a worker to perform the copy. For a closure, this worker thread will simply block until the collection of the environment has been completed. However, this will not in any way hinder the collection. In essence, the worker threads for types and values are already acting as worker threads for the environment. Defining additional worker threads for the environment may improve the efficiency of the collection, but they will not affect the correctness of the algorithm. We will prove that a single environment thread is sufficient in Chapter 10.

The server thread V^i for collecting environments is defined in Rule 9.20. The thread is activated when it receives an environment $cn^i_V\ ?\ (\Delta_1)$ along its local channel. The environment is decomposed into sub-environments which are in turn decomposed into pointers and locations. These pointers and locations are collected by sending them to the appropriate P and L servers. A new environment Δ_2 is constructed from the results of the collections, and returned $\underline{cn}^i_V\ !\ (\Delta_2)$ as the final result. We note that the sub-environments are collected in parallel, again removing an unnecessary bottleneck from the sequential algorithm. We will now define the collection of the stacks XS and RS.

$$
\begin{array}{l}
t_1 = cn_V^i \ ?\ (\Delta_1) \qquad \Delta_1 = (\mathit{TE}_1,\ \mathit{VE}_1,\ \mathit{DE}_1,\ \mathit{CE}_1) \\
\mathit{TE}_1 = \{\alpha^1 \ \mapsto\ p_1^1,\ \ldots,\ \alpha^k \ \mapsto\ p_1^k\} \\
t_2 = P^{num(p_1^1)} \ !\ (p_1^1)\ ?\ (p_2^1) \parallel \cdots \parallel P^{num(p_1^k)} \ !\ (p_1^k)\ ?\ (p_2^k) \\
\mathit{TE}_2 = \{\alpha^1 \ \mapsto\ p_2^1,\ \ldots,\ \alpha^k \ \mapsto\ p_2^k\} \\
\mathit{VE}_1 = \{x^1 \ \mapsto\ (l_1^1,\ p_3^1),\ \ldots,\ x^l \ \mapsto\ (l_1^l,\ p_3^l)\} \\
t_3 = P^{num(p_3^1)} \ !\ (p_3^1)\ ?\ (p_4^1) \parallel \cdots \parallel P^{num(p_3^l)} \ !\ (p_3^l)\ ?\ (p_4^l) \\
t_4 = L^{num(l_1^1)} \ !\ (l_1^1,\ p_3^1)\ ?\ (l_2^1,\ p_4^1) \parallel \cdots \parallel L^{num(l_1^l)} \ !\ (l_1^l,\ p_4^l)\ ?\ (l_2^l,\ p_4^l) \\
\mathit{VE}_2 = \{x^1 \ \mapsto\ (l_2^1,\ p_4^1),\ \ldots,\ x^l \ \mapsto\ (l_2^l,\ p_4^l)\} \\
\mathit{DE}_1 = \{tn^1 \ \mapsto\ p_5^1,\ \ldots,\ tn^m \ \mapsto\ p_5^m\} \\
t_5 = P^{num(p_5^1)} \ !\ (p_5^1)\ ?\ (p_6^1) \parallel \cdots \parallel P^{num(p_5^k)} \ !\ (p_5^k)\ ?\ (p_6^k) \\
\mathit{DE}_2 = \{tn^1 \ \mapsto\ p_6^1,\ \ldots,\ tn^m \ \mapsto\ p_6^m\} \\
\mathit{CE}_1 = \{con^1 \ \mapsto\ p_7^1,\ \ldots,\ con^n \ \mapsto\ p_7^n\} \\
t_6 = P^{num(p_7^1)} \ !\ (p_7^1)\ ?\ (p_8^1) \parallel \cdots \parallel P^{num(p_7^k)} \ !\ (p_7^k)\ ?\ (p_8^k) \\
\mathit{CE}_2 = \{con^1 \ \mapsto\ p_8^1,\ \ldots,\ con^n \ \mapsto\ p_8^n\} \\
\Delta_2 = (\mathit{TE}_2,\ \mathit{VE}_2,\ \mathit{DE}_2,\ \mathit{CE}_2) \qquad t_7 = \underline{cn}_V^i \ !\ (\Delta_2) \\
\hline
(\pi^1,\ \ldots,\ (\mathit{Hf}^i,\ \mathit{Ht}^i,\ \mathit{GT}^i,\ V \uplus \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k) \xrightarrow{t_1 \| t_2 \| t_3 \| t_4 \| t_5 \| t_6 \| t_7}_{gc} \\
\quad (\pi^1,\ \ldots,\ (\mathit{Hf}^i,\ \mathit{Ht}^i,\ \mathit{GT}^i,\ V \uplus \overline{\overline{\overline{T^i}}}),\ \ldots,\ \pi^k)
\end{array}
\tag{9.20}
$$

A stack is similar to an environment in that it contains only references to data, rather than actual data items. In principle, a stack worker thread is not required for the same reasons that a worker thread is not required when collecting an environment. However, a stack may contain pointers (p), value/type pairs ($l,\ p$) and environments (Δ) in any combination. Thus, it is natural to use a recursive definition for garbage collecting the stack, as we did in Chapter 6. However, the style of server thread which we have previously defined does not lend itself easily to recursive definitions.

In order to overcome this problem we use an approach which roughly approximates combination of both server and worker threads. We define a thread of the form $W_K(S_1,\ S_2)$, with three cases; one for each of the different kinds of stack item. In each case, we attempt to garbage collect the item at the top of the stack S_1 and place the result in S_2. These rules will cooperate to garbage collect the stack S_1, and will terminate when S_1 is empty and S_2 contains the collected stack. This solves the problem of how to collect the stack recursively using threads, but it does not solve the problem of how to fetch and return the stack over the communication channels. Therefore, we define a server thread K^i which fetches the stack and initialises the W_K threads with S_2 being empty. However, we now note that this server thread cannot simply restart once W_K has been initialised, since it is required to return a result on the communication channel, but this result will not be available until W_K has finished. Consequently, we suspend the server K^i when W_K is created and define an additional case for W_K which matches an empty stack S_1. In this final case, we send the result S_2 along the return channel and restart the server K^i. We define the server K^i and worker $W_K(S_1,\ S_2)$ threads for the stacks below:

$$\frac{t = cn^i_K\ ?\ (S_1)}{\begin{array}{l}(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ K^i \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \overset{t}{\Rightarrow}_{gc}\\ \quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ W_K(S_1,\ ()) \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k)\end{array}} \tag{9.21}$$

Comment: (Rule 9.21) The stack server thread K^i is activated when it receives a stack $cn^i_K\ ?\ (S_1)$ along its local channel. A worker thread $W_K(S_1,\ ())$ is created to copy the stack and the server is suspended.

$$\frac{t = P^{num(p_1)}\ !\ p_1\ ?\ p_2}{\begin{array}{l}(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ W_K(p_1 \cdot S_1,\ S_2,\ i) \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \overset{t}{\Rightarrow}_{gc}\\ \quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ W_K(S_1,\ S_2 \cdot p_2) \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k)\end{array}} \tag{9.22}$$

$$\frac{t1 = P^{num(p_1)}\ !\ p_1\ ?\ p_2 \qquad t_2 = L^{num(l_1)}\ !\ (l_1,\ p_1)\ ?\ (l_2,\ p_2)}{\begin{array}{l}(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ W_K((l_1,\ p_1) \cdot S_1,\ S_2) \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \xRightarrow{t_1\ ;\ t_2}_{gc}\\ \quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ W_K(S_1,\ S_2 \cdot (l_2,\ p_2)) \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k)\end{array}} \tag{9.23}$$

$$\frac{t = V^i\ !\ \Delta_1\ ?\ \Delta_2}{\begin{array}{l}(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ W_K(\Delta_1 \cdot S_1,\ S_2) \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \overset{t}{\Rightarrow}_{gc}\\ \quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ W_K(S_1,\ S_2 \cdot \Delta_2) \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k)\end{array}} \tag{9.24}$$

$$\frac{}{\begin{array}{l}(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ W_K(ty \cdot S_1,\ S_2) \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \overset{\epsilon}{\Rightarrow}_{gc}\\ \quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ W_K(S_1,\ S_2 \cdot ty) \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k)\end{array}} \tag{9.25}$$

$$\frac{}{\begin{array}{l}(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ W_K((ty,\ val) \cdot S_1,\ S_2) \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \overset{\epsilon}{\Rightarrow}_{gc}\\ \quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ W_K(S_1,\ S_2 \cdot (ty,\ val)) \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k)\end{array}} \tag{9.26}$$

Comment: (Rules 9.22, 9.23, 9.24, 9.25, and 9.26) These rules are the stack worker threads $WK(S_1,\ S_2)$. Each attempts to copy the item at the top of S_1 onto S_2 before restarting. Note that the item is placed at the bottom of S_2.

$$\frac{t = \underline{cn}^i_K\ !\ (S_2)}{\begin{array}{l}(\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ W_K((),\ S_2) \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \overset{t}{\Rightarrow}_{gc}\\ \quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i,\ K^i \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k)\end{array}} \tag{9.27}$$

Comment: (Rules 9.27) This final worker thread matches an empty stack S_1. The stack S_2 is returned along the channel $\underline{cn}^i_K\ !\ (S_2)$ and the server K^i is restarted.

9.1.4 Evaluation with Garbage Collection

We have now defined all of the threads required for distributed garbage collection. It therefore remains for us to address the initiation and termination of the collection. As stated, earlier, our distributed garbage collection algorithm requires a global synchronisation of all of the evaluation processes Π. Rule 9.28 defines the initialisation of a garbage collection process π from an evaluation process Π. This operation should be applied to every Π_1^i, where $0 < i \leq k$.

The set of threads $\overline{\overline{\overline{\mathcal{T}_1^i}}}$ contains the state of the evaluation at the point which the garbage collection is initiated. This state provides the roots for the garbage collection. These roots are all collected in parallel, as defined by the trace t_1. Garbage collection begins with an empty *to* heap Ht^i, an empty list of tables GT^i, and a multi-set of server threads $\{V_1^i \uplus K_1^i \uplus L_1^i \uplus P_1^i\}$. The trace t_2 describes all the interactions which occur during the collection.

At the conclusion of the garbage collection, we reconstruct the evaluation process Π_2^i from threads $\overline{\overline{\overline{\mathcal{T}_2^i}}}$ from the collected versions of all the roots. These roots will now contain only references to the *to* heap H_2^i. Finally, the *from* heap H_1^i is discarded and normal evaluation is resumed. Note that the contents of the cache Hc_1^i is cleared by garbage collection, since the placement of the data in the heap will have changed.

$$\frac{\begin{array}{l} \overline{\overline{\overline{\mathcal{T}_1^i}}} = \{(\Delta_1^1,\ XS_1^1,\ RS_1^1),\ \ldots,\ (\Delta_1^j,\ XS_1^j,\ RS_1^j)\} \\ t_1\ =\ V^i\ !\ \Delta_1^1\ ?\ \Delta_2^1 \parallel \cdots \parallel V^i\ !\ \Delta_1^j\ ?\ \Delta_2^j \parallel \\ \qquad K^i\ !\ XS_1^1\ ?\ XS_2^1 \parallel \cdots \parallel K^i\ !\ XS_1^j\ ?\ XS_2^j \parallel \\ \qquad K^i\ !\ RS_1^1\ ?\ RS_2^1 \parallel \cdots \parallel K^i\ !\ RS_1^j\ ?\ RS_2^j \\ (\pi_1^1,\ \ldots,\ (H_1^i,\ \emptyset,\ (\emptyset,\ \emptyset,\ \emptyset,\ \emptyset),\ \{V_1^i \uplus K_1^i \uplus L_1^i \uplus P_1^i\}),\ \ldots,\ \pi_1^k) \overset{t_2}{\Rightarrow}_{gc} \\ \quad (\pi_2^1,\ \ldots,\ (H_1^i,\ H_2^i,\ GT^i,\ \{V_2^i \uplus K_2^i \uplus L_2^i \uplus P_2^i\}),\ \ldots,\ \pi_2^k) \\ \overline{\overline{\overline{\mathcal{T}_2^i}}} = \{(\Delta_2^1,\ XS_2^1,\ RS_2^1),\ \ldots,\ (\Delta_2^j,\ XS_2^j,\ RS_2^j)\} \end{array}}{(\Pi_1^1,\ \ldots,\ (H_1^i,\ Hc_1^i,\ \overline{\overline{\overline{\mathcal{T}_1^i}}}),\ \ldots,\ \Pi_1^k) \overset{t_1 \| t_2}{\Longrightarrow} (\Pi_2^1,\ \ldots,\ (H_2^i,\ \emptyset,\ \overline{\overline{\overline{\mathcal{T}_2^i}}}),\ \ldots,\ \Pi_2^k)} \tag{9.28}$$

Rule 9.29 is applied when the set of threads in process Π_1^i is empty. There may still be data in the local heap that is referenced by threads on other processes, so the L^i and P^i servers must be initialised to perform the collection.

$$\frac{\begin{array}{l} (\pi_1^1,\ \ldots,\ (H_1^i,\ \emptyset,\ (\emptyset,\ \emptyset,\ \emptyset,\ \emptyset),\ \{L_1^i \uplus P_1^i\}),\ \ldots,\ \pi_1^k) \overset{t}{\Rightarrow}_{gc} \\ \quad (\pi_2^1,\ \ldots,\ (H_1^i,\ H_2^i,\ GT^i,\ \{L_2^i \uplus P_2^i\}),\ \ldots,\ \pi_2^k) \end{array}}{(\Pi_1^1,\ \ldots,\ (H_1^i,\ Hc_1^i,\ \emptyset),\ \ldots,\ \Pi_1^k) \overset{t}{\Rightarrow} (\Pi_2^1,\ \ldots,\ (H_2^i,\ \emptyset,\ \emptyset),\ \ldots,\ \Pi_2^k)} \tag{9.29}$$

9.2 Code Replacement

The threading and communication extensions for distributed evaluation in Dynamic ML were defined in Chapter 8. As we saw, all of these extensions were defined at the core-language level. Consequently, there are no changes required to the Dynamic ML modules system for distributed evaluation. Therefore, the modules interpretation scheme from Chapter 7 can be used unchanged for distributed code-replacement. The definition of distributed code-replacement which follows, is purely an extension of the distributed garbage collection algorithm.

As in Chapter 7, we do not provide a definition of the meta-level at which separate-compilation is performed, and code-replacement is introduced. Instead, we take as our starting point the pair $(P,\ R)$ of a program P which contains the new definitions for use after replacement, and the program R which contains the functions to perform the code-replacement, i.e. the `Install` structures.

We begin by defining the initialisation of code-replacement. The main activity occurs within the first thread on the first process Π^1. Recall from Chapter 8 that this is also where the evaluation process begins, and where all the threads originate. The algorithm begins with Rule 9.30 by evaluating the program R to obtain the replacement map RM, exactly as in Chapter 7. However, in the distributed case, all of the other processes in the garbage collection now also need a copy of this replacement map. Consequently, we define a new channel cn_R for each process in the system, which we use to broadcast the replacement map.

$$\frac{\begin{array}{l}
\overline{\overline{\overline{\mathcal{T}_1^1}}} = \{(\Delta_1^1,\ XS_1^1,\ RS_1^1),\ \ldots,\ (\Delta_1^j,\ XS_1^j,\ RS_1^j)\} \\
((H_1^1,\ Hc_1^1,\ \{(\Delta_1^1,\ XS_1^1,\ RS_1^1,\ R)\}),\ \ldots,\ \Pi^k) \overset{t_1}{\Rightarrow} \\
\quad ((H_2^1,\ Hc_2^1,\ \{(\Delta_2^1,\ XS_2^1,\ (l_1,\ p_1) \cdot RS_2^1)),\ \ldots,\ \Pi^k) \\
H_2^1(l_1) = \overline{l_2}^{\,n} \qquad H_2^1(p_2^1) = p_3^1 \rightarrow p_4^1\ \cdots\ H_2^1(p_2^n) = p_3^n \rightarrow p_4^n \\
RM = \{p_3^1\ \mapsto\ (l_2^1,\ p_4^1),\ \ldots,\ p_3^n\ \mapsto\ (l_2^n,\ p_4^n)\} \\
t_2\ =\ (cn_R^2\ !\ RM \parallel \cdots \parallel cn_R^i\ !\ RM)\ ; \\
\qquad V^i\ !\ \Delta_2^1\ ?\ \Delta_3^1 \parallel V^i\ !\ \Delta_1^2\ ?\ \Delta_2^2 \cdots \parallel V^i\ !\ \Delta_1^j\ ?\ \Delta_2^j \parallel \\
\qquad K^i\ !\ XS_2^1\ ?\ XS_3^1 \parallel K^i\ !\ XS_1^2\ ?\ XS_2^2 \parallel \cdots \parallel K^i\ !\ XS_1^j\ ?\ XS_2^j \parallel \\
\qquad K^i\ !\ RS_2^1\ ?\ RS_3^1 \parallel K^i\ !\ RS_1^2\ ?\ RS_2^2 \parallel \cdots \parallel K^i\ !\ RS_1^j\ ?\ RS_2^j \\
((H_2^1,\ \emptyset,\ (RM,\ \emptyset,\ \emptyset,\ \emptyset),\ \{V_1^1 \uplus K_1^1 \uplus L_1^1 \uplus P_1^1\}),\ \ldots,\ \pi_1^k) \overset{t_3}{\Rightarrow}_{gc} \\
\quad ((H_2^1,\ H_3^1,\ GT^1,\ \{V_2^1 \uplus K_2^1 \uplus L_2^1 \uplus P_2^1\}),\ \ldots,\ \pi_2^k) \\
((H_3^1,\ \emptyset,\ \{(\Delta_3^1,\ XS_3^1,\ RS_3^1,\ P)\}),\ \ldots,\ \Pi^k) \overset{t_4}{\Rightarrow} \\
\quad ((H_4^1,\ Hc_3^1,\ \{(\Delta_4^1,\ XS_4^1,\ (l_3,\ p_5) \cdot RS_4^1)),\ \ldots,\ \Pi^k) \\
\overline{\overline{\overline{\mathcal{T}_2^1}}} = \{(\Delta_4^1,\ XS_4^1,\ RS_4^1),\ (\Delta_2^2,\ XS_2^2,\ RS_2^2),\ \ldots,\ (\Delta_2^j,\ XS_2^j,\ RS_2^j)\}
\end{array}}{((H_1^1,\ Hc_1^1,\ \overline{\overline{\overline{\mathcal{T}_1^1}}}),\ \ldots,\ \Pi^k) \xRightarrow{t_1\ ;\ t_2 \| t_3\ ;\ t_4}_{rep} ((H_4^1,\ Hc_3^1,\ \overline{\overline{\overline{\mathcal{T}_2^1}}}),\ \ldots,\ \Pi^k)} \tag{9.30}$$

The initialisation of code-replacement for the remaining processes $\Pi^2, \ldots, \Pi^k$ is defined in Rule 9.31, or Rule 9.32 when the process has an empty set of threads. The processes are initially blocked until they receive the replacement map on their local channel $\underline{cn}^i_R \mathrel{?} RM$. Once all of the processes have received a copy of the replacement map, a distributed garbage collection, extended with the code-replacement operations which we shortly define, is performed. As before, the effect will be to update all of the live data to the new representations, and discard any unreachable data. Upon successful completion, the program P is evaluated on process Π^1 and evaluation resumes on every process.

$$\frac{\begin{array}{l}\overline{\overline{\mathcal{T}^i_1}} = \{(\Delta^1_1,\ XS^1_1,\ RS^1_1),\ \ldots,\ (\Delta^j_1,\ XS^j_1,\ RS^j_1)\} \\ t_1 \ = \ \underline{cn}^i_R \mathrel{?} RM \ ; \ V^i \mathrel{!} \Delta^1_1 \mathrel{?} \Delta^1_2 \parallel \cdots \parallel V^i \mathrel{!} \Delta^j_1 \mathrel{?} \Delta^j_2 \parallel \\ \qquad K^i \mathrel{!} XS^1_1 \mathrel{?} XS^1_2 \parallel \cdots \parallel K^i \mathrel{!} XS^j_1 \mathrel{?} XS^j_2 \parallel \\ \qquad K^i \mathrel{!} RS^1_1 \mathrel{?} RS^1_2 \parallel \cdots \parallel K^i \mathrel{!} RS^j_1 \mathrel{?} RS^j_2 \\ (\pi^1_1,\ \ldots,\ (H^i_1,\ \emptyset,\ (RM,\ \emptyset,\ \emptyset,\ \emptyset),\ \{V^i_1 \uplus K^i_1 \uplus L^i_1 \uplus P^i_1\}),\ \ldots,\ \pi^k_1) \overset{t_2}{\Rightarrow}_{gc} \\ \quad (\pi^1_2,\ \ldots,\ (H^i_1,\ H^i_2,\ GT^i,\ \{V^i_2 \uplus K^i_2 \uplus L^i_2 \uplus P^i_2\}),\ \ldots,\ \pi^k_2) \\ \overline{\overline{\mathcal{T}^i_2}} = \{(\Delta^1_2,\ XS^1_2,\ RS^1_2),\ \ldots,\ (\Delta^j_2,\ XS^j_2,\ RS^j_2)\}\end{array}}{(\Pi^2,\ \ldots,\ (H^i_1,\ Hc^i_1,\ \overline{\overline{\mathcal{T}^i_1}}),\ \ldots,\ \Pi^k) \overset{t_1 \parallel t_2}{\Longrightarrow}_{rep} (\Pi^2,\ \ldots,\ (H^i_2,\ \emptyset,\ \overline{\overline{\mathcal{T}^i_2}}),\ \ldots,\ \Pi^k)} \tag{9.31}$$

$$\frac{\begin{array}{l}t_1 \ = \ \underline{cn}^i_R \mathrel{?} RM \\ (\pi^1_1,\ \ldots,\ (H^i_1,\ \emptyset,\ (RM,\ \emptyset,\ \emptyset,\ \emptyset),\ \{L^i_1 \uplus P^i_1\}),\ \ldots,\ \pi^k_1) \overset{t_2}{\Rightarrow}_{gc} \\ \quad (\pi^1_2,\ \ldots,\ (H^i_1,\ H^i_2,\ GT^i,\ \{L^i_2 \uplus P^i_2\}),\ \ldots,\ \pi^k_2)\end{array}}{(\Pi^2,\ \ldots,\ (H^i_1,\ Hc^i_1,\ \emptyset),\ \ldots,\ \Pi^k) \overset{t_1\ ;\ t_2}{\Longrightarrow}_{rep} (\Pi^2,\ \ldots,\ (H^i_2,\ \emptyset,\ \emptyset),\ \ldots,\ \Pi^k)} \tag{9.32}$$

The distributed code-replacement algorithm can also be *rolled-back* if an exception is thrown during the operation. In this case, all of the processes in the collection are suspended and evaluation resumes with the previous machine state. This is possible as the *from* heaps, containing the old data, are preserved until the end of replacement. Rule 9.33 is a modified version of Rule 9.31 which illustrates the roll-back of a single process in the system. The remaining roll-back rules, for the process Π^1 and for a process with an empty set of threads, are very similar and are not explicitly defined here.

$$\frac{\begin{array}{l}\overline{\overline{\mathcal{T}^i_1}} = \{(\Delta^1_1,\ XS^1_1,\ RS^1_1),\ \ldots,\ (\Delta^j_1,\ XS^j_1,\ RS^j_1)\} \\ t_1 \ = \ \underline{cn}^i_R \mathrel{?} RM \ ; \ V^i \mathrel{!} \Delta^1_1 \mathrel{?} \Delta^1_2 \parallel \cdots \parallel V^i \mathrel{!} \Delta^j_1 \mathrel{?} \Delta^j_2 \parallel \\ \qquad K^i \mathrel{!} XS^1_1 \mathrel{?} XS^1_2 \parallel \cdots \parallel K^i \mathrel{!} XS^j_1 \mathrel{?} XS^j_2 \parallel \\ \qquad K^i \mathrel{!} RS^1_1 \mathrel{?} RS^1_2 \parallel \cdots \parallel K^i \mathrel{!} RS^j_1 \mathrel{?} RS^j_2 \\ (\pi^1_1,\ \ldots,\ (H^i_1,\ \emptyset,\ (RM,\ \emptyset,\ \emptyset,\ \emptyset),\ \{V^i_1 \uplus K^i_1 \uplus L^i_1 \uplus P^i_1\}),\ \ldots,\ \pi^k_1) \overset{t_2}{\Rightarrow}_{gc} \\ \quad (\pi^1_2,\ \ldots,\ (H^i_1,\ H^i_2,\ GT^i,\ \mathbf{halt}\ \{V^i_2 \uplus K^i_2 \uplus L^i_2 \uplus P^i_2\}),\ \ldots,\ \pi^k_2)\end{array}}{(\Pi^2,\ \ldots,\ (H^i_1,\ Hc^i_1,\ \overline{\overline{\mathcal{T}^i_1}}),\ \ldots,\ \Pi^k) \overset{t_1 \parallel t_2}{\Longrightarrow}_{rep} (\Pi^2,\ \ldots,\ (H^i_1,\ Hc^i_1,\ \overline{\overline{\mathcal{T}^i_1}}),\ \ldots,\ \Pi^k)} \tag{9.33}$$

9.2.1 Replacement of Types and Values

To complete the definition of distributed code-replacement, we must make a number of minor additions to the server threads in the distributed garbage collector. There are just three additional rules required, which we will now present.

Rule 9.34 extends the type server thread P^i with code-replacement. An incoming pointer p_1 is compared with the replacement map, using the *rmatch* function defined in Chapter 7. Provided that the type referenced by the pointer has not already been collected, and that a suitable match p_2 is found in RM^i, a replacement is performed. This is achieved by garbage collecting collecting the replacement type p_3, in place of the old type p_1. A pointer p_4 is reserved in the *to* heap to hold the result, and a worker thread $W_P(p_3,\ p_4)$ is generated to perform the copy. The forwarding table is updated to reflect the replacement $PF^i[p_1 \ \mapsto\ p_4]$, and the result p_4 is returned along the server channel.

$$\frac{\begin{array}{l} t_1 = cn^i_P\ ?\ (p_1) \\ GT^i_1 = (RM^i,\ PF^i,\ LF^i,\ TE^i) \qquad p_1 \notin \mathrm{Dom}\ PF^i \\ Hf_1 \vdash rmatch(p_1,\ RM^i) = p_2 \qquad RM(p_2) = (l_1,\ p_3) \\ Ht^i \uparrow p_4 \qquad GT^i_2 = (RM^i,\ PF^i[p_1 \ \mapsto\ p_4],\ LF^i,\ TE^i) \\ t_2 = \underline{cn^i_P}\ !\ (p_4) \end{array}}{\begin{array}{l} (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i_1,\ P^i \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \xrightarrow{t_1\ ;\ t_2}_{rep} \\ \quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i_2,\ P^i \uplus W_P(p_3,\ p_4) \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \end{array}} \tag{9.34}$$

The value server L^i extended with code-replacement is defined by Rule 9.35. The rule begins when an incoming value $(l_1,\ p_1)$ is received. Assuming that the value has not been collected, and that a match is found in the replacement map, a closure must also be evaluated to update the value. This closure is obtained from the replacement map, and the result if its evaluation $(l_3,\ p_4)$ is substituted for the old value by garbage collection.

$$\frac{\begin{array}{l} t_1 = cn^i_L\ ?\ (l_1,\ p_1) \\ GT^i_1 = (RM^i,\ PF^i,\ LF^i,\ TE^i) \qquad l_1 \notin \mathrm{Dom}\ LF^1 \\ Hf_1 \vdash rmatch(p_1,\ RM^i) = p_2 \qquad RM(p_2) = (l_2,\ p_3) \\ Hf_1(l_2) = \langle\!\langle \Delta_1,\ \{x\},\ E \rangle\!\rangle \\ (\Pi^1_1,\ \ldots,\ (Hf^i_1,\ Hc^i_1,\ \{(\Delta_1[x \ \mapsto\ (l_1,\ p_1)],\ (),\ (),\ E)\},\ \ldots,\ \Pi^k_1) \overset{t_2}{\Rightarrow} \\ \quad (\Pi^1_2,\ \ldots,\ (Hf^i_2,\ Hc^i_2,\ \{(\Delta_2,\ XS,\ (l_3,\ p_4) \cdot RS)\}),\ \ldots,\ \Pi^k_2) \\ Ht^i \uparrow l_4 \qquad GT^i_2 = (RM^i,\ PF^i,\ LF^i[l_1 \ \mapsto\ l_4],\ TE^i) \\ t_3 = \underline{cn^i_L}\ !\ (l_4,\ PF^i(p_1)) \end{array}}{\begin{array}{l} (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i_1,\ L^i \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \xrightarrow{t_1\ ;\ t_2\ ;\ t_3}_{rep} \\ \quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i_2,\ L^i \uplus W_L(l_3,\ p_4,\ l_4,\ p_5) \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \end{array}} \tag{9.35}$$

Finally, the evaluation of the replacement closure in L^i may generate an uncaught exception. This should result in the code-replacement being aborted and the machine state rolled-back to the configuration prior to the start of replacement. This is initiated by performing a **halt** within the abstract machine, as shown in Rule 9.36.

$$\frac{\begin{array}{l} t_1 = cn^i_L \; ? \; (l_1,\ p_1) \\ GT^i_1 = (RM^i,\ PF^i,\ LF^i,\ TE^i) \qquad l_1 \notin \mathrm{Dom}\ LF^1 \\ Hf_1 \vdash rmatch(p_1,\ RM^i) = p_2 \qquad RM(p_2) = (l_2,\ p_3) \\ Hf_1(l_2) = \langle\!\langle \Delta_1,\ \{x\},\ E \rangle\!\rangle \\ (\Pi^1_1,\ \ldots,\ (Hf^i_1,\ Hc^i_1,\ \{(\Delta_1[x \mapsto (l_1,\ p_1)],\ (),\ (),\ E)\},\ \ldots,\ \Pi^k_1) \overset{t_2}{\Rightarrow} \\ \quad (\Pi^1_2,\ \ldots,\ (Hf^i_2,\ Hc^i_2,\ \mathbf{halt}\ \{(\Delta_2,\ XS,\ RS)\}),\ \ldots,\ \Pi^k_2) \end{array}}{\begin{array}{l} (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i_1,\ L^i \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \xRightarrow{t_1\ ;\ t_2}_{rep} \\ \quad (\pi^1,\ \ldots,\ (Hf^i,\ Ht^i,\ GT^i_2,\ \mathbf{halt}\ L^i \uplus \overline{\overline{T^i}}),\ \ldots,\ \pi^k) \end{array}} \tag{9.36}$$

9.3 Optimisation

In this chapter we have defined the basic algorithms for distributed garbage collection and replacement. The abstract machine model has also provided a formal definition for the mechanisms of the LEMMA memory interface. However, it is important to note that many improvements to these algorithms can be made, which may prove to be highly beneficial in an implementation.

A clear optimisation to the algorithm would be the preservation of the contents of the heap cache Hc across garbage collections. At present, our algorithm entirely clears the cache at each collection, as the items in the cache will have changed locations. In an implementation, this will result in a significant drop in performance while the cache is re-populated. We note that it is possible to partially preserve the contents of the cache, if we know the new locations of the data. This information is present in the forwarding tables PF and LF, and may therefore be used to reorder the cache. Nonetheless, this strategy would require careful definition to avoid interference with code-replacement.

Another cache optimisation concerns mutable data items. Recall from Chapter 8, that only immutable data is cached in our definition, since this kind of data does not change and does not require a coherency algorithm. We note that this approach will produce poor performance if a program contains more than a few mutable values. A number of algorithms which perform coherency checks on mutables are detailed in [SM94]. A further optimisation which allows data items to migrate between processes in the LEMMA interface is described in [MS95a].

Chapter 10

Verification

In this chapter we will use the SPIN [Hol97b] model checker to perform a mechanical verification of our distributed garbage collection algorithm. The devastating effects of an incorrect garbage collection algorithm on program execution make it a prime candidate for mechanical verification techniques. Such techniques provide a degree of safety that is unattainable by testing. Mechanical verification is particularly relevant in concurrent and distributed systems where errors are hard to trigger, for example, an error caused by a particular interleaving of processes. The properties of our algorithm that we are seeking to validate are:

- **Progress** - absence of deadlock (or live-lock) in the algorithm.
- **Correctness** - no corruption of data during collection.
- **Preservation** - no dangling pointers after collection.

SPIN is a generic verification system which is designed for proving the correctness of process interactions in concurrent systems. It accepts design specifications in its own language PROMELA (PROcess MEta-LAnguage), and verifies correctness claims specified as a Linear Temporal Logic (LTL) formula. SPIN has been in development for several years and includes a large number of techniques for improving the efficiency of the model checking, for example, partial-order reduction, state-compression, and on-the-fly verification [Hol97a].

In this chapter we present a brief overview of the model checking technique, with particular reference to the SPIN model checker. We then present a detailed specification of our garbage collection algorithm in PROMELA. Finally, we define a test-bed for verifying the properties shown above, and present our results and conclusions.

10.1 Model Checking

Model checking is a technique for automatically verifying properties of finite-state concurrent systems. A model checker normally performs an exhaustive search of the state space of a system to determine if a particular property holds. Given sufficient resources, the procedure will always terminate with a yes/no answer. This makes the model checking technique of significant practical value as a verification tool. The restriction to finite-state systems may appear to be a major disadvantage. However, there exist techniques for restricting infinite-state systems to specific finite instances, e.g. [McM99]. There have also been attempts to combine theorem proving with model checking, e.g. [DF95, RSS95]. However, both of these techniques are unnecessary here, since our distributed garbage collection algorithm is a finite-state system.

A model checker operates on an abstract model of a system. Clearly, the definition of this model requires some thought in order to ensure that the correct level of detail is provided. For the majority of systems it is not sufficient simply to consider the input and output behaviour. The model must capture the *states* of the system, and the acceptable *transitions* between these states. This is precisely the kind of description which our abstract machine formalism already provides. Thus, in principle, we could define a model checking system directly for our abstract machine definition. However, this would clearly not be a particularly profitable activity, as we would have to construct the model checker from scratch. Rather, we will adopt the representation of an existing model checker, and perform a straightforward translation from our model.

The underlying representation used in the SPIN model checker, among others, is a state-transition graph called a *Kripke structure*. This graph may be defined formally as a four tuple $M = (S,\ S_0,\ R,\ L)$, where:

1. S is a finite set of states, and $S_0 \subseteq S$ is a set of initial states.

2. $R \subseteq S \times S$ is a reachability relation. Note that R must be *total*, i.e. for every state $s \in S$, these is a state $s' \in S$ such that $R(s,\ s')$ is defined.

3. $L : S \rightarrow \overline{\overline{A}}$ is a function that labels each state with the set of atomic propositions A which are true in that state.

Different restrictions can be imposed on R for defining models which have properties such as asynchronous or synchronous evaluation [CGP99].

Figure 10.1 defines an example Kripke structure for a very simple system. We will use this structure as a running example in our explanation of the model checking process.

1. $S = \{s_0,\ s_1,\ s_2\} \qquad S_0 = \{s_0\}$
2. $R = \{(s_0,\ s_1),\ (s_0,\ s_2),\ (s_1,\ s_0),\ (s_1,\ s_2),\ (s_2,\ s_2)\}$
3. $L = (s_0\ \mapsto\ \{a,\ b\},\ s_1\ \mapsto\ \{b,\ c\},\ s_2\ \mapsto\ \{c\})$

Figure 10.1: *Example Kripke Structure.*

Given a system defined as a Kripke structure, we want to specify correctness properties of the system for the model checker to verify. The common choices of representation are temporal logic formulae and the μ-calculus. The one that is of importance here is the temporal formulae representation as this approach is used in the SPIN model checker. We will not discuss the μ-calculus further except to say that it is another formalism, based on fixed-points, which can encode many temporal logics.

Conceptually, temporal formulae define properties of *computational trees.* The tree is formed by designating a state in the Kripke structure as the initial state, which will form the root, and then unrolling the remaining structure into an infinite tree. Figure 10.2 shows a graphical representation of our example Kripke structure, and its corresponding computation tree.

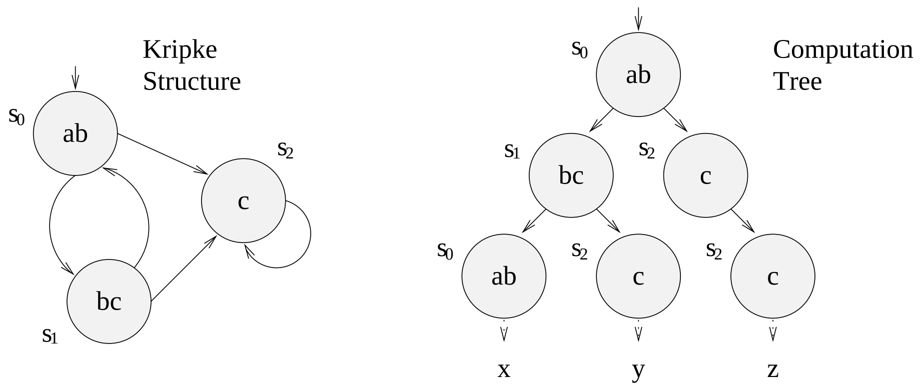


Figure 10.2: *Unrolling to a Computation Tree.*

We can think of each branch of this computation tree as a *path* in a Kripke structure, where each path corresponds to a different simulation of the system. Formally, a path is defined as an infinite sequence of states $\sigma = s_0\ s_1\ s_2\ \cdots$, such that for every $i \geq 0$, $(s_i,\ s_{i+1}) \in R$.

The view of a Kripke structure as a computation tree exposes two different kinds of temporal formulae: *linear-time* and *branching-time*. In a linear-time logic we express properties over all possible paths of the tree, and in a branching-time logic we express properties over a single path of the tree. There are many different logics which fall into these categories. However, from a model-checking perspective, the commonly used logics are the linear temporal logic (LTL) and computation tree logic (CTL) respectively. Both of these logics share the same basic syntactic operators: always $\Box$, eventually $\Diamond$, and until $\mathcal{U}$. However, the semantics of the operators differs depending on the logic in question. Figure 10.3 illustrates the difference between LTL and CTL with reference to a computation tree. The operators $\Box$ and $\Diamond$ can be thought of as temporal variants of $\forall$ and $\exists$ respectively. The (strong) until property $p\,\mathcal{U}\,q$ states that p remains true until at least q becomes true. In LTL we must satisfy the property along every path in the tree, while in CTL is is sufficient that the property is satisfied along one particular path.

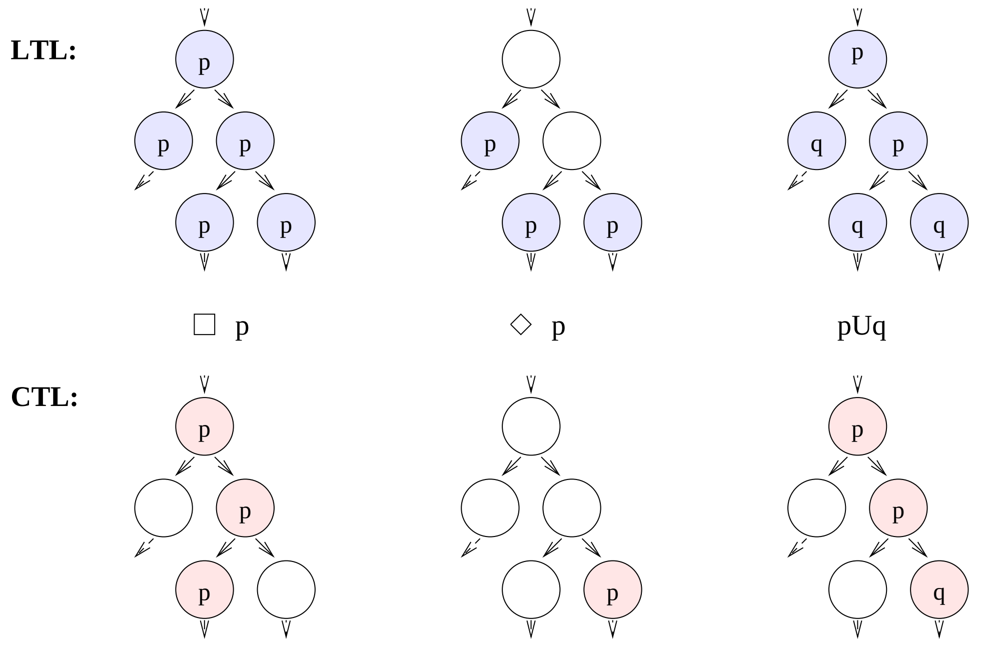


Figure 10.3: *Linear and Branching Temporal Logics.*

LTL and CTL are both used in real model checking systems. The SPIN model checker uses the former, while the SMV model checker [McM93] uses the latter. It is worth noting that LTL and CTL are not equivalent, e.g. the LTL property $\Diamond\Box p$ cannot be expressed in CTL. The differences between these logics are discussed in detail in [Var01]. We will limit the remainder of our discussion to LTL as we will be using the SPIN model checker.

The formal syntax of the propositional linear temporal logic (LTL) is defined in Figure 10.4. LTL is powerful enough to express a wide range of properties. For example, the formula $\Box(s \rightarrow \Diamond r)$ is a representation of the property that every message sent s is eventually received r. Note that we generally verify the negation of the property in question as this yields a more efficient search.

Formula	f	$::=$	$\top \mid \bot$	(boolean true and false)
		$\mid$	p	(proposition)
		$\mid$	$\neg f$	(negation)
		$\mid$	$\Box f$	(always)
		$\mid$	$\Diamond f$	(eventually)
		$\mid$	$f_1 \wedge f_2$	(conjunction)
		$\mid$	$f_1 \vee f_2$	(disjunction)
		$\mid$	$f_1 \rightarrow f_2$	(implication)
		$\mid$	$f_1 \leftrightarrow f_2$	(equivalence)
		$\mid$	$f_1 \, \mathcal{U} \, f_2$	(strong until)

Figure 10.4: *LTL Abstract Syntax.*

To summarise, we have now defined an appropriate representation for the system model (Kripke structures) and a language for expressing properties (LTL formulae). We have seen how LTL formulae relate to Kripke structures through the use of computation trees. It therefore remains to complete the picture by formally stating the model checking problem. Given a Kripke structure M and an LTL formula f, we find the set of all states that satisfy f. We say that the system satisfies the specification if all the initial states are in this set:

$$S_0 \subseteq \{s \in S \mid M, s \models f\}$$

Model checking is therefore a very straightforward technique utilising known results from temporal logic. The difficulty in performing model-checking, and the focus of a great deal of research in model checking is the provision of efficient algorithms. At the core of most modern model checking systems is a representation of the computation tree as an Ordered Binary Decision Diagram (OBDD) [CGP99]. Model checking is then performed by a depth-first (or similar) traversal of this structure. Advanced techniques have been developed for reducing the state space of the search and for generating the OBDD structures on-the-fly, though in general it is not necessary to understand the underlying optimisations in order to use a particular model checking system.

10.2 System Model

The SPIN model checker performs the same kind of LTL model checking as defined in the previous section, though SPIN contains a large number of refinements and optimisations over the basic algorithm. However, we will not discuss the mechanics of model checking in further detail. In the remainder of this chapter we will concentrate on using model checking to verify our distributed garbage collector.

The first step in the verification is the construction of an appropriate system model. The underlying framework for modelling in SPIN is the Kripke structure, though this is well hidden underneath its own process meta-language PROMELA, a CCS-like language containing processes and channels, similar to the primitives in Dynamic ML (which are also CCS-based). SPIN translates the PROMELA language into Kripke structures, through a (loose) mapping of processes to states and channels to transitions.

The syntax of the PROMELA resembles C, as it is designed to appeal to engineers, though this similarity is rather misleading as many of the C constructs are missing or have a different semantics. In the remainder of this section, we define a specification of the distributed garbage collection algorithm in PROMELA. A basic knowledge of the C language is assumed in the following explanation. The differences between PROMELA and C will be explained as they are encountered.

Before proceeding, we must first resolve a number of issues concerning the modelling of a distributed system in PROMELA. In Chapter 9 we define the state as a set of processes $\overline{\overline{\pi}}$, where each process π refers to an individual processor with a local memory, e.g. a workstation or a supercomputer node. Each process evaluates a set of threads $\overline{\overline{\overline{T}}}$ in parallel. The number of processes in the system is fixed, while the number of threads can vary dynamically. Since we are using a distributed shared memory model, each thread T may directly access the memory of any process π^i. By contrast, PROMELA is not a distributed system: it only provides one level of abstraction: *threads*. Note that a thread is named a `proctype` in PROMELA. All of the threads in PROMELA are executed together and cannot be nested. Thus, in order to model our system, we make the following simplification. We remove the distinction between processes and execute all of the threads together. The threads are suitably renamed to prevent name-space collisions. The memory of the distributed system is modelled by a global array which is directly accessible by all of the threads. Since we are not concerned with communication delays or failures here, this model is adequate for our purposes.

10.2.1 Shared Memory

An important aspect of the model is the *memory* upon which the garbage collection operations will be performed. Our PROMELA specification of the memory, and other data structures that will be used in our model, is given in Figure 10.6. For clarity, we divide the address space into four arrays of equal size, for types `t` and values `v`: `tfrom`, `vfrom`, `tto`, and `vto`. The first two arrays represent the global *from* space, and the remaining two arrays represent the global *to* space. These arrays will be partitioned into local *from* and *to* spaces. There are two arrays for each space as we require a separate type heap and value heap for tag-free collection. The size of these arrays are calculated from the equivalent number of processes in our distributed system. We define the total number of such processes by the constant `PROCESSES`. For simplicity, we fix the size of every heap on each process to be the constant `HEAPSIZE`. Thus, the size of each array, `TOTAL`, is obtained by multiplying the heap size by the number of processes. For practical reasons the `HEAPSIZE` and `PROCESSES` will typically range between 1 and 4 in our model. The rationale behind these values will be explained in more detail in Section 10.2.3. A diagram of our model of the address space is presented in Figure 10.5.

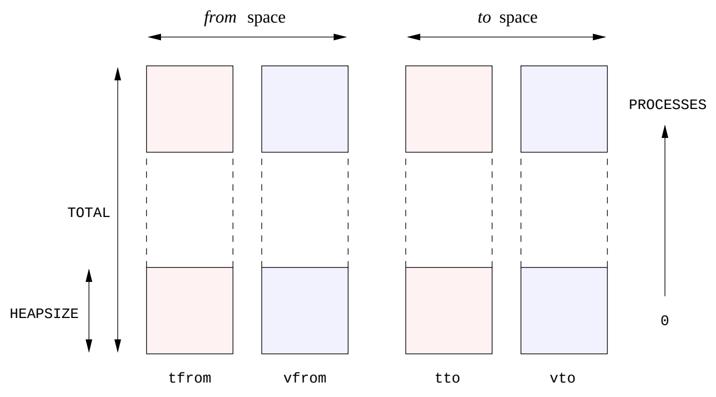


Figure 10.5: *Address Space Model.*

We define pointers into the type-heap as variables prefixed with `p`, e.g. `p1, p2`, and locations in the value-heap as variables prefixed with `l`. We have defined an additional type `Pair` to express a pairing of a value-location `l` and a type-pointer `p`. As in C, types in PROMELA are declared using the `typedef` construct. However, PROMELA does not have explicit pointers and so `l` and `p` are simply indices into the respective arrays. In order to conserve memory, we use `byte` for representing the pointers as the range of heap indices should be bounded by `TOTAL`, which will be far less than the maximum value of 255.

```
#define HEAPSIZE  3  /* Size of Each Heap */
#define PROCESSES 3  /* Number of Processes */
#define TOTAL     9  /* Total Memory Size (HEAPSIZE * PROCESSES) */

/* Environment */
typedef Env{ byte l[TOTAL], p[TOTAL], size };

/* Stack */
typedef Stak{ byte l[TOTAL], p[TOTAL], size };

/* Type */
typedef Type { byte p[TOTAL], size };

/* Value */
typedef Value { byte l[TOTAL], val ;
                Env e };

/* Type/Value Pointer */
typedef Pair { byte l, p };

/* Heaps */
Type tfrom[TOTAL], tto[TOTAL];
Value vfrom[TOTAL], vto[TOTAL];

/* Program Environment and Stacks */
Env efrom[PROCESSES], eto[PROCESSES];
Stak sfrom[PROCESSES], sto[PROCESSES];

/* Forwarding Tables */
byte pforward[TOTAL], lforward[TOTAL];
#define NULL 255                              /* Special NULL Value */

/* Allocation Pointers in the to-space */
byte pto[PROCESSES], lto[PROCESSES];

/* Communication Channels */
chan pschan[PROCESSES] = [0] of {byte};     /* Type Server Channels */
chan prchan[PROCESSES] = [0] of {byte};
chan lschan[PROCESSES] = [0] of {Pair};    /* Value Server Channels */
chan lrchan[PROCESSES] = [0] of {Pair};
chan eschan[PROCESSES] = [0] of {Env};      /* Environment Channels */
chan erchan[PROCESSES] = [0] of {Env};
chan sschan[PROCESSES] = [0] of {Stak};           /* Stack Channels */
chan srchan[PROCESSES] = [0] of {Stak};
```

Figure 10.6: *Data Structures for Verification.*

The environment `Env` is defined as a pair of arrays: one for type-pointers, and one for value-locations. In the worst-case, this environment could contain a pointer to everything in the address space, hence the size of the arrays is `TOTAL`. An additional variable `size` tracks the actual number of entries in the environment. PROMELA does not permit recursive definitions as these would lead to infinite state models. Hence, the stack `Stak` is defined in exactly the same way as the environment.

The garbage collection algorithm is defined for a range of different types and values. We could provide a separate PROMELA type definition for each different type and value. However, for the purpose of modelling the algorithm, we are only concerned with the pointers they contain. Thus, a `Type` is simply defined as an array of pointers `p`. In the worst case, the memory is occupied by a single tuple which fills the entire address space. Therefore, the size of the pointer array is `TOTAL`. Once again, the variable `size` tracks the actual number of pointers. Values are represented in a similar manner. We define a `Value` as an array of locations `l` of size `TOTAL`. The number of entries in the array is determined by the variable `size` in the associated type. In order to allow the representation of closures, each value also contains an environment `e`.

Before defining the threads which we will use in garbage collection, it is necessary to define a number of structures upon which they rely. These structures are also given in Figure 10.6. The garbage collection algorithm performs compaction of the data objects. Thus, objects in the *from* space will reside at different locations from those in the *to* space. Arrays of forwarding pointers `pforward` and `lforward` are maintained to track the copying of objects, corresponding to the finite maps *PF* and *LF* in Chapter 9. Each array index corresponds to an address in the *from* space, and the array value contains a pointer to the corresponding address in the *to* space, or a special `NULL` value if there is none. During copying, the next free location in the *to* space is determined linearly using the arrays `pto` and `lto`. There is a separate entry in these arrays for each process. Each server has two associated communication channels: one for sending and one for receiving. The arrays `pschannel` and `prchannel` contain the channels for sending and receiving pointers to/from the type-servers respectively. Similar arrays are defined for the value servers, environment servers, and stack servers. Communication channels are defined as variables of type `chan`. They are initialised by an expression of the form `[N] of {type}`, where `N` is the size of the buffer and `type` is the type of data that can be passed along the channel. For our purposes the buffer size is always 0, which corresponds to an unbuffered blocking channel.

10.2.2 Garbage Collection Threads

Having defined an appropriate model of the memory, we proceed with a model of the actual garbage collection algorithm. We begin with a definition of the threads for collecting types and values. There is a one-to-one mapping between the garbage collection threads T of Chapter 9 and PROMELA threads (`proctype`). It is clear that the number of server threads for types and values will each be equal to the constant `PROCESSES` defined earlier, i.e. one server for every process. Note that although we are not explicitly modelling garbage collection processes π in PROMELA, we still use the equivalent number of processes in the construction of the model.

Each server thread will operate on a separate area of the heap in the range $i *$ `HEAPSIZE` to $(i + 1) *$ `HEAPSIZE`, where i is the number of the server, $0 \leq i <$ `PROCESSES`. Given a pointer `p` or location `l` as an index in the range 0 to `TOTAL`, the appropriate server i may be calculated by dividing the value of the pointer by `HEAPSIZE`. This is equivalent to the *num* function defined in Chapter 8. Before presenting the thread definitions, we will briefly describe the basics of PROMELA statements and threads.

In PROMELA there is no difference between conditionals and statements. The execution of each statement is conditional on its executability; statements are either *executable* or *blocked*. For example, the statement `(a == b)` can only be executed when it holds. If it does not, then the thread containing the statement is blocked until a later time when the condition does hold. The `do` loop is the only kind of iterative statement in PROMELA:

```
do
  :: guard1 -> sequence1
  :: guard2 -> sequence2
  :: ...
  :: else -> sequence
od
```

The body of a loop contains a sequence of execution statements, each preceded by a double colon. The first statement in the sequence is called a *guard*, as the remainder of the sequence can only be executed if the guard is executable. Only one of the sequences may be executed on each iteration of the loop. In the case where more than one guard is executable, a non-deterministic choice is made between the executable sequences. If none of the guards are executable, then the thread containing the loop is blocked until at least one of the guards becomes executable. The special guard `else` is selected if all of the other guards are blocked. A loop can only be exited by a `break` statement.

The semicolon in PROMELA is a statement *separator*, not a statement terminator as in C. It separates the sequential execution of statements, and is equivalent to sequential composition in Dynamic ML. Consequently, there is no semicolon after the final statement in a block of code. An arrow `->` is also a statement separator and is identical in meaning to the semicolon. However, the arrow is generally used to indicate a causal relation between two statements.

The only unit of execution in PROMELA is the thread. There are no language constructs for declaring functions, predicates or procedures, although these can be simulated by threads. PROMELA threads are declared using `proctype` expressions and executed using `run` statements. All of the threads run concurrently and asynchronously with respect to one another. Threads can take argument values when they are created, analogous to call-by-value parameter passing of arguments to functions. Values can be returned from threads through shared global variables or messages passed along channels. Initialisation of a program, analogous to the `main` function in C, is performed via a special `init` thread which is automatically evaluated at the start.

The server threads for types and values are defined in Figure 10.7. The `TypeServer` thread immediately enters a `do` loop where it is blocked until a pointer `p1` is received on channel `pschannel[process]`. Note that the notation for sending `!` and receiving `?` along channels is identical to the equivalent operations in Dynamic ML. The `if` statement is used in the body of the type server to distinguish between types that have already been copied and types that have not. An `if` statement performs identically to a single iteration of a `do` loop. The first guard corresponds to a type that has not been copied. In this case, a new pointer `p2` is obtained from the array `pto[process]`. Allocations are performed linearly in the *to* heap, and so the value of `pto[process]` is subsequently incremented by one. The forwarding table `pforward` is updated to provide a mapping from `p1` to `p2`, and a `TypeWorker(p1, p2)` thread is generated to perform the copy. The `else` guard corresponds to a type that has already been copied. In this case, the pointer `p2` is simply obtained from the forwarding table `pforward[p1]`. After the evaluation of the `if` statement, the pointer `p2` is returned along the channel `prchannel[process]` and the server enters a new iteration. The `end:` statement is an end-state label; it indicates to the model checker that the start of the loop is a valid termination point. The `atomic` statement is an optimisation which reduces the number of states. The definition of the `ValueServer` thread is very similar to the type-server. However, as the copying of values is done in the tag-free style, pairs of value-locations and type-pointers are used instead.

```
proctype TypeServer(byte process)              /* Type Server Thread */
{
  byte p1, p2;
  end: do
      :: pschan[process] ? p1 ->               /* Obtain Next Pointer */
           atomic
           {
             if
               :: (pforward[p1] == NULL) ->
                    p2 = pto[process];         /* Reserve New Pointer */
                    pto[process] =
                      pto[process] + 1;
                    pforward[p1] = p2;       /* Update Forward Table */
                    run TypeWorker(p1, p2)   /* Create Worker Thread */
               :: else ->
                    p2 = pforward[p1]      /* Pointer Already Copied */
             fi
           };
           prchan[process] ! p2             /* Return Copied Pointer */
    od                                              /* Restart Server */
}

proctype ValueServer(byte process)            /* Value Server Thread */
{
  Pair lp1, lp2;
  end: do
      :: lschan[process] ? lp1 ->                 /* Obtain Next Pair */
           atomic
           {
             if
               :: (lforward[lp1.l] == NULL) ->
                    lp2.l = lto[process];
                    lto[process] =
                      lto[process] + 1;
                    lforward[lp1.l] = lp2.l;
                    run ValueWorker(lp1, lp2)
               :: else ->
                    lp2.l = lforward[lp1.l]
             fi;
             lp2.p = lp1.p;
           };
           lrchan[process] ! lp2               /* Return Copied Pair */
    od
}
```

Figure 10.7: *Type and Value Server Threads.*

The worker threads for types and values are defined in Figure 10.8. As we have defined both type and values as arrays of pointers, these threads closely correspond to the collection of tuples in Chapter 9. We can effectively simulate a special constant as a tuple with no pointers, a constructor type as a tuple with one pointer, and a function type as a tuple with two pointers. The `TypeWorker` thread is passed the pointers `p1` and `p2` as arguments. The thread creates a new type in the *to* space referenced by `p2` by setting the `size` of the tuple. The type referenced by `p1` is then recursively copied into `p2` by garbage collecting all of the pointers it contains. A single `do` loop iterates through these pointers. Each pointer `p3` is sent to the appropriate server, and the result `p4` is entered into the type at `p2`. As stated earlier, the appropriate server thread is calculated by dividing the pointer value by `HEAPSIZE`. The worker terminates when all of the pointers have been collected. The `ValueWorker` thread is again very similar. A single `do` loop is used to iterate through all of the locations for garbage collection. However, note that the associated type `tfrom[lp1.p]` is used to determine the length of the value `vfrom[lp1.l]`, effectively simulating the process of tag-free collection. A value may contain a closure environment. When there are entries in this environment, it is collected by sending it to the local environment server.

We will now model the threads for collecting environments and stacks. As we have seen, the lack of recursion in PROMELA means that both environments and stacks share the same definition. However, we define rather different, though broadly equivalent, threads for their collection. The reason for this relates to the differences between the corresponding threads in Chapter 9. The `EnvServer` thread is defined in Figure 10.9. The thread is blocked until an environment is sent to the thread along the channel `eschan[process]`. The environment `e2` will hold the result of the collection. Note that passing an environment across a channel creates a new copy of the environment, i.e. message passing is done by-value rather than by-reference. Hence, we can reuse the environment `e2` on each iteration of the server. The `size` of `e2` is determined by the size of the environment `e1` received on the input channel. The body of the thread is simply a `do` loop which iterates through each entry in the environment `e1` in turn. For each entry, the type pointer is collected first, by sending it to the appropriate `TypeServer`. The value location is then collected by constructing a pair `lp1` and sending it to the corresponding `ValueServer`. The results are stored the environment `e2`, which is returned along the channel `erchan[process]` at the end.

```
proctype TypeWorker(byte p1, p2)              /* Type Worker Thread */
{
  byte p3, p4, count, process;

  tto[p2].size = tfrom[p1].size;                    /* Create New Type */
  count = 0;
  do
    :: (count < tfrom[p1].size) ->     /* Step Through Every Pointer */
         p3 = tfrom[p1].p[count];
         process = p3 / HEAPSIZE;         /* Calculate Server Number */
         pschan[process] ! p3;        /* Send Pointer To Type Server */
         prchan[process] ? p4;         /* Wait For Reply From Server */
         tto[p2].p[count] = p4;                     /* Update New Type */
         count ++
    :: else -> break
  od
}

proctype ValueWorker(Pair lp1; Pair lp2)      /* Value Worker Thread */
{
  Pair lp3, lp4;
  byte count, process, local;

  vto[lp2.l].val = vfrom[lp1.l].val;               /* Create New Value */
  count = 0;
  do
    :: (count < tfrom[lp1.p].size) ->        /* Collect All Pointers */
         lp3.p = tfrom[lp1.p].p[count];
         lp3.l = vfrom[lp1.l].l[count];
         process = lp3.l / HEAPSIZE;
         lschan[process] ! lp3;
         lrchan[process] ? lp4;
         vto[lp2.l].l[count] = lp4.l;
         count ++
    :: else -> break
  od;
  if
    :: (vfrom[lp1.l].e.size > 0) ->           /* Collect Environment */
         loc = lp1.l / HEAPSIZE;
         eschan[loc] ! vfrom[lp1.l].e;         /* Send To Env Server */
         erchan[loc] ? vto[lp2.l].e         /* Reply From Env Server */
    :: else -> vto[lp2.l].e.size = 0            /* Empty Environment */
  fi
}
```

Figure 10.8: *Type and Value Worker Threads.*

```
proctype EnvServer(byte process)        /* Environment Server Thread */
{
  Pair lp1, lp2;
  Env e1, e2;
  byte count;
  byte process2;

  end:
    do
      :: eschan[process] ? e1 ->              /* Obtain Environment */
           e2.size = e1.size;
           count = 0;
           do                               /* Collect All Pointers */
             :: (count < e1.size) ->
                  process2 = e1.p[count] / HEAPSIZE;
                  pschan[process2] ! e1.p[count];
                  prchan[process2] ? e2.p[count];
                  lp1.p = e1.p[count];
                  lp1.l = e1.l[count];
                  process2 = e1.l[count] / HEAPSIZE;
                  lschan[process2] ! lp1;
                  lrchan[process2] ? lp2;
                  e2.l[count] = lp2.l;
                  count ++
             :: else -> break
           od;
           erchan[process] ! e2            /* Return New Environment */
    od
}
```

Figure 10.9: *Environment Thread.*

The `StackServer` is defined in Figure 10.10. We have attempted to remain as close as possible to the definition in Chapter 9. The collection begins when the `StackServer` thread receives stack `s1` on its input channel `sschan[process]`. A new stack `s2` is initialised to hold the result, and a `StackWorker` thread is created to start the copy. Note that arrays cannot be passed as arguments to processes in PROMELA. Hence, a temporary channel `temp` is used to pass the stacks to the worker. Once the worker thread has received the stacks, the server terminates. Each item in the stack is collected in turn by a separate `StackWorker` thread. The `if` statement in the thread is used to determine if the end of the stack has been reached. The first guard corresponds to the case when there are still items on the stack `s1`. The top item is collected, and a new `StackWorker` thread is created for the remainder of the stack (by passing an incremented `count`). Stacks `s1` and `s2` are passed across the channel `temp2` to the new thread and the old thread terminates. The `else` guard holds when the end of the stack is reached. The collected stack `s2` is returned along the channel `srchan[process]` and the `StackServer` thread is restarted.

```
proctype StackServer(byte process)            /* Stack Server Thread */
{
  Stak s1, s2;
  chan temp = [0] of {Stak};         /* Temporary Unbuffered Channel */

  end:
    sschan[process] ? s1 ->
      run StackWorker(process, 0, temp);     /* Create Worker Thread */
      temp ! s1; temp ! s2                  /* Pass Stacks To Worker */
}                                              /* Terminate Server */

proctype StackWorker(byte process1; byte count; chan temp1)
{
  Pair lp1, lp2;
  byte process2;
  Stak s1, s2;
  chan temp2 = [0] of {Stak};

  temp1 ? s1; temp1 ? s2;       /* Receive Stacks From Server/Worker */
  if
    :: (count < s1.size) ->
          process2 = s1.p[count] / HEAPSIZE;
          pschan[process2] ! s1.p[count];            /* Collect Type */
          prchan[process2] ? s2.p[count];
          lp1.p = s1.p[count];
          lp1.l = s1.l[count];
          process2 = s1.l[count] / HEAPSIZE;
          lschan[process2] ! lp1;                   /* Collect Value */
          lrchan[process2] ? lp2;
          s2.l[count] = lp2.l;
          run StackWorker(process1, count + 1, temp2);
          temp2 ! s1; temp2 ! s2
    :: else ->
          s2.size = s1.size;                         /* End Of Stack */
          srchan[process1] ! s2;          /* Return Collected Stack */
          run StackServer(process1)               /* Restart Server */
  fi
}
```

Figure 10.10: *Stack Threads.*

10.2.3 User Program

We have now provided a specification in PROMELA for the garbage collection threads of Chapter 9. However, these threads cannot simply be verified in isolation. Rather, our verification strategy is based on observing the behaviour of the collector on a range of different complex data structures. Consequently, we require an additional model, corresponding to a user program evaluated prior to the collection, to generate this data.

One possible approach would be a model implementation, or partial implementation, of the evaluation semantics for Dynamic ML defined in Chapter 8. Heap data for verifying the collector could then be generated by the evaluation of a number of programs within this model. However, this approach would require a significant programming effort. Furthermore, it would only show that the collector was correct for the supplied examples, and would not show correctness in the general case.

In order to provide a sufficient level of confidence in the verification, we will adopt an approach promoted by [Hav99] and [Bru97]. We restrict the size of the heap to a reasonably small value, and then entirely fill the heap with tuples of random length containing random pointers. We arrange the state space of the model such that the model checker will explore all possible heap configurations for the given heap size. Figure 10.11 illustrates all of the possible acyclic tuples for a heap size of 3. In our model, we will also permit cyclic structures, which yields approximately 100 different configurations. Due to the exponential increase in configurations with heap size, we use a heap size no larger than 4 to perform our verification. While it may be argued that the collector may contain errors which only become apparent with larger heap sizes, we believe that this is not the case. We have found that accidental errors in our specification were rapidly exposed with a heap size of just 2 (and 2 processes). It is not possible to obtain absolute confidence in the general case without resorting to a theorem proving approach, e.g. [Jac98]. However, this would require a significantly more complex specification, and we would lose the fully-automated property of model-checking.

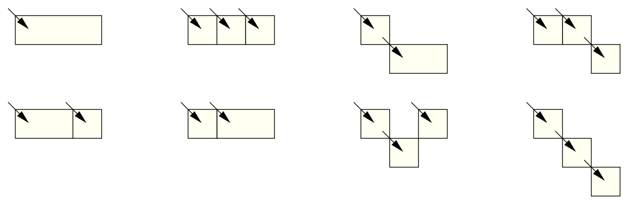


Figure 10.11: *Acyclic Tuple Allocation (length 3)*

One of the main problems with the model checking technique is that even moderately sized systems, such as the one here, rapidly lead to a very large state-space that precludes exhaustive verification. This problem occurs in our model even when we restrict the size of the heap and the number of garbage collection processes to the range of 1 to 4. However, the situation is far from hopeless. The SPIN model checker addresses this problem with a facility called *bit-state hashing* [Hol98]. This technique uses complex hashing functions to map the state space of the model onto the available memory. The effect of this facility is to provide a high coverage approximation of the results of an exhaustive verification using considerably less memory. The coverage can be further improved by performing multiple runs with statistically independent hashing functions. The SPIN model checker provides a percentage estimate of the coverage of the search space at the end of each run. We quote these figures in our results presented later in the chapter.

The PROMELA specification of the user program is given in Figure 10.12 as a new thread called `Mutator`. The construction of this thread is very straightforward. The thread essentially behaves like a greedy program and completely fills the memory with random data. A heap pointer `hp` references the next available heap location. This pointer is used to track the allocation of tuples in the heap. Every iteration of the outermost `do` loop allocates a new tuple on the heap. The loop (and the thread) terminates when the heap space has been entirely exhausted. There are three stages involved in the allocation of every tuple.

The first step is the determination of the length of the tuple, given by the variable `count1`. Zero length tuples are also permitted, corresponding to special constants. These tuples consume one heap location and contain no additional pointers. The tuple length is obtained by non-determinism: the second `do` loop contains two guards; the first guard increments the length of the tuple by one, and the second guard `:: break` terminates the loop. Since both of these guards will hold (unless the heap is full), a non-deterministic choice will be made between them, producing a random tuple length. The maximum tuple length is bounded by the remaining space in the heap, i.e. `hp + 1 + count1 < TOTAL`. Once the length of the tuple has been set, the `size` field is set to the value of `count1`, and the `val` field is updated with the tuple number.

The second step is the population of the tuples with pointers. This step is performed in concert with the allocation of the tuple. Each position in the tuple is allocated a pointer, given by the variable `count2`. Non-determinism is used again in the selection of random locations within the heap.

```
proctype Mutator()                            /* User Program Thread */
{
  byte hp, val, process, count1, count2;
  atomic
  {
    hp = 0; val = 0;
    do
      :: (hp < TOTAL) ->                       /* Check If Heap Full */
           count1 = 0;
           do                                /* Choose Tuple Length */
             :: (hp + 1 + count1 < TOTAL) ->
                  count2 = 0;              /* Choose Random Pointer */
                  do
                    :: (count2 < TOTAL - 1) -> count2 ++
                    :: break
                  od;
                  tfrom[hp].p[count1] = count2;    /* Populate Tuple */
                  vfrom[hp].l[count1] = count2;
                  count1 ++
             :: break
           od;
           tfrom[hp].size = count1;          /* Set Final Tuple Size */
           vfrom[hp].val = val;
           val ++;
           process = 0;           /* Enter Into Environment or Stack */
           do
             :: (process < PROCESSES) ->
                  if
                    :: efrom[process].p[efrom[process].size] = hp;
                       efrom[process].l[efrom[process].size] = hp;
                       efrom[process].size ++
                    :: sfrom[process].p[efrom[process].size] = hp;
                       sfrom[process].l[efrom[process].size] = hp;
                       sfrom[process].size ++
                    :: skip
                  fi;
                process ++
             :: else -> break
           od;
           hp = hp + 1 + count1            /* Increment Heap Pointer */
      :: else -> break
    od
  }
}
```

Figure 10.12: *User Program Model.*

In the final step we iterate through all of the processes in the system. For each process, the tuple is placed in either the environment or stack of the process, or simply ignored. These three cases are handled by the non-deterministic `if` statement (note the absence of guards). A tuple may be ignored by every process, in which case it becomes garbage to be removed by the collector.

It is worth noting that the model checker does not operate directly on the non-deterministic model that we have defined. Instead, a deterministic tree is constructed by the unrolling process described earlier in this chapter. For the `Mutator` thread, a tree will be constructed with a separate path for every possible heap configuration.

10.2.4 Initialisation

The initial thread `init` is defined in Figure 10.13. This thread initiates the three sequential phases of the model: the `Mutator` thread is evaluated to generate the initial data, a garbage collection is then performed, and lastly a `Validator` thread is evaluated to compare the initial and collected data. These phases are synchronised by the loop `do :: timeout -> break od;` as we do not want collection to begin before the data has been generated, or validation to be performed before collection is complete. The `timeout` guard succeeds only when there are no executable threads remaining. We could use message passing for synchronisation, but we would need to track all of the worker threads, resulting in a considerably more complex solution.

The garbage collection phase begins by an initialisation of the forwarding arrays `pforward` and `lforward` to `NULL` and the allocation arrays `lto` and `pto` to the beginning of the local *to* spaces. A type server, value server, environment server, and stack server are also generated for each process. We pass each thread the process number `process` to distinguish between them. As we saw earlier, this number is used by each thread to reference its local communication channels and heaps.

The initialisation stage is done atomically as we do not want collection to begin until all the server threads are in place. Collection then begins by sending all the environments and stacks to their respective servers. This results in a cascade of communication between the servers, which in turn results in the generation of worker threads and so on. Eventually, new environments and stacks are generated and returned as messages. The collection is finished when all the threads have terminated, or all the threads are blocked if an error has occurred.

```
init                                        /* Initialisation Thread */
{
  byte process, heap;
  run Mutator();  do :: timeout -> break od;           /* Fill Heaps */
  atomic
  {
    process = 0;
    do
      :: (process < PROCESSES) ->
           heap = 0;
           do                                     /* Forwarding Tables */
             :: (heap < HEAPSIZE) ->
                  pforward[heap + (process * HEAPSIZE)] = NULL;
                  lforward[heap + (process * HEAPSIZE)] = NULL;
                  heap ++
             :: else -> break
           od;
           lto[process] = process * HEAPSIZE;   /* Allocation Tables */
           pto[process] = process * HEAPSIZE;
           run TypeServer(process);                 /* Start Servers */
           run ValueServer(process);
           run EnvServer(process);
           run StackServer(process);
           process ++
      :: else -> break
    od
  };
  process = 0;
  do
    :: (process < PROCESSES) ->
         eschan[process] ! efrom[process];   /* Collect Environments */
         sschan[process] ! sfrom[process];         /* Collect Stacks */
         process ++
    :: else -> break;
  od;
  process = 0;
  do
    :: (process < PROCESSES) ->
         erchan[process] ? eto[process];         /* New Environments */
         srchan[process] ? sto[process];               /* New Stacks */
         process ++
    :: else -> break;
  od; do :: timeout -> break od; run Validator()       /* Validation */
}
```

Figure 10.13: *Garbage Collection Introduction.*

10.3 Validation

The properties of the garbage collection algorithm that we wish to validate were stated at the beginning of the chapter, namely: *progress*, *preservation*, and *correctness*. We will now consider each of these properties in turn, and identify an appropriate validation technique.

The progress condition is the most straightforward to validate. Given that progress is a requirement in almost every concurrent system, the SPIN model checker automatically ensures this property by default. Every thread has one or more associated *end* states, which denote the valid termination points. The final state of a thread is implicitly an end state. Additional end states may be explicitly defined through the insertion of `end:` labels, as in the type and value server threads of Figure 10.7. The progress condition states that every thread eventually reaches a valid end state. This can be expressed as the following LTL formula, where `end1` is the end state for the first thread, and `end2` is the end state for the second thread, etc:

$$\Box(\ \Diamond(\texttt{end1} \wedge \texttt{end2} \wedge \texttt{end3} \wedge \cdots))$$

Progress is violated if a thread fails to reach an end state, e.g. a worker thread fails to receive a reply from a server and remains blocked. The model checker will iterate through every possible evaluation for a given heap size. Hence, this property will ensure that progress is always preserved.

As we saw earlier in the chapter, the underlying Kripke structures in SPIN are concealed by the PROMELA language. Similarly, while it is possible to state explicit LTL formulae for SPIN to verify, it is considerably more convenient to state properties using PROMELA *assertions* and *never-claims*. Assertions are statements that a particular property is always true at a particular point. Never claims state that a particular property will never be true at a particular point, for example:

```
assert(even % 2 == 0);
never(odd % 2 == 0);
```

An assertion of the form `assert(cond)` is equivalent to an LTL formula of the form $\Box(s \wedge \texttt{cond})$, where s is the state in which the condition `cond` should hold. A never-claim is equivalent to the negation $\Box(s \wedge \neg\, \texttt{cond})$. The advantage of placing these statements directly inside the PROMELA program, instead of defining separate formula, is that we do not have to explicitly define the state s, and we can embed assertions inside loops and other constructs.

The validation of the correctness and preservation properties requires a significantly more complex approach than the progress condition. We define an additional `Validator` thread for this purpose, shown in Figure 10.14. Correctness and preservation are validated together in this thread, since these properties are closely related.

The high-level view of our validation strategy is a direct comparison of the graphs of the data items in the *from* heap, with their corresponding graphs in the *to* heap. This is achieved in our thread by a depth-first graph-search algorithm which operates on each item in the *from* heap in-turn. The search begins with the root pointer of the item in the environment (or stack). At each step of the search, a comparison is performed with the counterpart in the *to* heap, obtained from the forwarding tables. The search terminates when all of the pointers have been followed. The behaviour of this algorithm is essentially the same as the sequential garbage collection algorithm of Chapter 6. Each item has an associated type and value, which are validated as follows.

- The correctness property is enforced by comparing the `size` fields of the types, and the `val` fields of the values. Recall that the `val` field of each thread was tagged with a different tuple number during allocation in Figure 10.12. Hence, only values which are identical will be matched. This check ensures that the data has not been corrupted during collection.

- The preservation property is validated by a comparison of each pointer in each type, and each location in each value. The entry in the forwarding table for the pointer or location in the *from* heap should be identical to the corresponding pointer or location in the *to* heap. If the entry in the forwarding table is `NULL`, or the resulting pointers are not equal, then an error will have occurred in the collection and there may be dangling pointers.

We will now outline the construction of our validation strategy in PROMELA. As stated above, we use a thread called `Validate` to perform this step. The appropriate point for the creation of this thread was shown in Figure 10.13. The absence of true recursion in the language led us to adopt a stack-based depth-first algorithm, using the array `stack`. This should not be confused with the `Stak` upon which garbage collection is performed.

The stack will contain `Pair` pointers to items which we have not yet validated. At each step, we pop a pair off the stack and examine the item it references. Any pointers contained within this item are subsequently pushed onto the stack. The algorithm terminates when the stack is empty. As we are dealing with data which has the form of an undirected graph, there may be cyclic structures. Therefore, we maintain a set of pointers which we have already validated, represented by the boolean array `pseen`. This array contains an entry for each pointer in the heap, where each entry is initially set to `false`. On encountering a pointer, the corresponding entry in `pseen` is set to `true`. Items which are set to `true` in the array are not pushed onto the `stack`. The algorithm is entirely deterministic, indicated by the statement `d_step` at the beginning of the thread. This allows SPIN to perform a number of optimisations on the model.

We validate each `process` in turn. A number of sanity checks are performed before the depth-first traversal is initiated. We check that the environment and stack for the process are the same size before and after collection. We also check that the number of types is equal to the number of values after collection, and that the size of the local heap is smaller than the maximum permitted heap size, i.e. we have not introduced any extra data during collection. These simple tests proved to be surprisingly effective at trapping accidental errors in our specification.

The main depth-first algorithm validates each entry in the environment in turn. The pointer `ep` references the current environment entry, i.e. root. We do not show the validation of the roots of the program stack in Figure 10.14, as the algorithm is identical to the validation of the environment. Another pointer `sp` contains the index of the top of the `stack`. The core of the algorithm is a `do` loop with two guards. The first guard `(sp == 0)` holds when the stack is empty; the next item in the environment is selected and validated, or the algorithm terminates if there are none. The second guard `(sp > 0)` holds when there are pointers on the stack; in this case, the stack is popped and the corresponding item validated. The validation process is essentially the same in both cases. We check correctness by a pair of assertions which compare the `val` and `size` fields:

```
assert(tfrom[loc1].size == tto[loc2].size);
assert(vfrom[loc3].val == vto[loc4].val);
```

The remaining assertions on the forwarding tables `pforward` and `lforward` check that pointers in the *from* heaps were actually copied to the correct positions in the *to* heaps. This effectively ensures the preservation property that there are no dangling pointers:

```
loc5 = tfrom[loc1].p[count1];
loc6 = vfrom[loc3].l[count1];
loc7 = tto[loc2].p[count1];
loc8 = vto[loc4].l[count1];
assert(pforward[loc5] == loc7);
assert(lforward[loc6] == loc8);
```

```
proctype Validator()                            /* Validator Thread */
{
  byte ep, sp, process, count1, count2;
  byte loc1, loc2, loc3, loc4, loc5, loc6, loc7, loc8;
  Pair stack[TOTAL];
  bool pseen[TOTAL];
  d_step
  {
    count1 = 0;
    do
      :: (count1 < TOTAL) ->
           pseen[count1] = false;   /* Initially All Pointers Unseen */
           count1 ++
      :: else -> break
    od;
    process = 0;
    do
      :: (process < PROCESSES) ->
           assert(efrom[process].size == eto[process].size);
           assert(sfrom[process].size == sto[process].size);
           assert(lto[process] == pto[process]);
           assert(pto[process] <= (process +1) * HEAPSIZE);
           ep = 0;
           sp = 0;
           do
             :: (sp == 0) ->              /* Empty Stack - Next Root */
                  if
                    :: (ep < efrom[process].size) ->
                         loc1 = efrom[process].p[ep];
                         loc2 = eto[process].p[ep];
                         loc3 = efrom[process].l[ep];
                         loc4 = eto[process].l[ep];
                         assert(pforward[loc1] == loc2);
                         assert(lforward[loc3] == loc4);
                         assert(tfrom[loc1].size == tto[loc2].size);
                         assert(vfrom[loc3].val == vto[loc4].val);
                         pseen[loc1] = loc2;
                         count1 = 0;
                         do
                           :: (count1 < tfrom[loc1].size) ->
                                loc5 = tfrom[loc1].p[count1];
                                loc6 = vfrom[loc3].l[count1];
                                loc7 = tto[loc2].p[count1];
                                loc8 = vto[loc4].l[count1];
                                assert(pforward[loc5] == loc7);
                                assert(lforward[loc6] == loc8);
```

```
                              if
                                :: (pseen[loc5] == false) ->
                                     pseen[loc5] = true;
                                     stack[sp].l = loc6;
                                     stack[sp].p = loc5;
                                     sp ++; count1 ++
                                :: else -> count1 ++
                              fi
                          :: else -> break
                        od;
                        ep ++
                    :: else -> break
                  fi
              :: (sp > 0) ->            /* Non Empty Stack - Pop Item */
                  sp --;
                  loc1 = stack[sp].p;
                  loc2 = stack[sp].l;
                  assert(pforward[loc1] != NULL);
                  assert(lforward[loc2] != NULL);
                  assert(tfrom[loc1].size == tto[pforward[loc1]].size);
                  assert(vfrom[loc2].val == vto[lforward[loc2]].val);
                  count1 = 0;
                  do
                    :: (count1 < tfrom[loc1].size) ->
                         loc3 = tfrom[loc1].p[count1];
                         loc4 = vfrom[loc2].l[count1];
                         loc5 = tto[pforward[loc1]].p[count1];
                         loc6 = vto[lforward[loc2]].l[count1];
                         assert(pforward[loc3] == loc5);
                         assert(lforward[loc4] == loc6);
                         if
                           :: (pseen[loc3] == false) ->
                                pseen[loc3] = true;
                                stack[sp].l = loc4;
                                stack[sp].p = loc3;
                                sp ++; count1 ++
                           :: else -> count1 ++
                         fi
                    :: else -> break
                  od
           od; process ++
       :: else -> break
     od
  }
}
```

Figure 10.14: *Consistency Checks.*

10.4 Results and Conclusions

We performed a model checking operation on our specification for the range of process numbers and heap sizes shown in Figure 10.15. The bit-state hashing technique was used throughout as none of these models were small enough for exhaustive verification, given the 512MB of memory that we had available. An estimate of the coverage of the search space is shown as the final column of the table. The case where only one process is used corresponds to the sequential algorithm. Although the numbers in the table appear to be rather small, we believe that they are adequate to demonstrate the correctness of the algorithm. As we showed earlier, a total heap size of just 3 results in over 100 different configurations. Furthermore, a system with just 3 processes will lead to a similarly large number of inter-leavings of evaluation. Consequently, it is almost certainly the case that any errors in the specification would become readily apparent in this situation. It was our experience that minor accidental errors in our specification led to an immediate violation of the assertions.

GC Processes	Local Heap Size	Total Heap Size	Estimated Coverage
1	3	3	100%
2	1	2	98%
2	2	4	74%
3	1	3	61%
4	1	4	42%

Figure 10.15: *Model Configurations.*

The original formalisation of the LEMMA distributed garbage collection algorithm appeared in [Wal99]. The model was carefully constructed, with reference to the actual LEMMA implementation, to avoid deadlocks in the algorithm. It was this formalisation that we chose as the starting point for our model checking process. This model is very similar to the one presented in Chapter 9, but only a single channel was used for communication between the server and worker threads. The model checking process immediately uncovered a problem with this approach, illustrated in Figure 10.16.

The problem occurs when two (or more) worker threads W1 and W2 attempt to communicate with a single server thread S along a single channel (step 1). It will always be the case that one thread will succeed and the other one will be blocked. In this case (step 2), thread W1 succeeds and thread W2 is blocked. However, we are now in a situation where thread W1 is attempting to receive a reply from S on the channel and W2 is still attempting to send a message on

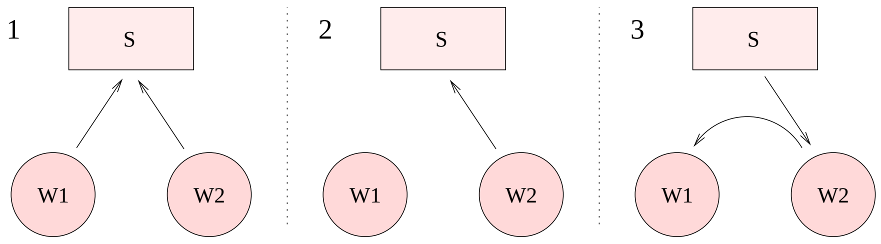


Figure 10.16: *Communication Error.*

the same channel. Thus, W1 actually receives the message from W2 instead of the reply from S, and the reply from S is incorrectly received by W2 (step 3). While this problem is obvious in retrospect, it is the kind of problem that is very difficult to detect with simple testing. The problem will only occurs infrequently in practice, due to timing issues. Furthermore, this problem does not generally cause a deadlock to occur, since all the communication operations are successful. Nonetheless, the mistake was readily uncovered by model checking. The solution to the problem is to use pairs of channels in the communication. The corrected specification is the one presented in Chapter 9. With this revised specification, no other errors were detected in the algorithm.

10.5 Limitations of Model Checking

It should be noted that the model checking technique, as with all verification strategies, has a number of inherent limitations. We have already discussed the restrictions on our model arising from the finite nature of model checking, in particular the need to consider heaps of fixed sizes. We note that this limitation can be overcome by theorem proving, though this would likely come at the expense of an automated verification. Other limitations of our verification relate to the properties that we have chosen to validate. There is considerable difficulty in ensuring that the chosen properties are sufficient to validate all of the necessary behaviours of the model. For example, in our current model we check that the structure of the heap is preserved but we do not explicitly check that the garbage is discarded. The model checker cannot assist in this respect since the required properties are largely dependent on the domain in question.

The most serious limitation of the model checking technique arguably relates to its shortcomings regarding the expression of certain real-world conditions. For example, the eventually $\diamond$ temporal operator does not say anything about the time interval involved. As a result, we cannot say anything about the time requirements of our algorithm, which may take a very long time to complete. Another such condition relates to our communication channels. We currently assume that there are no communication delays or lost messages, and that messages are passed instantaneously. If we allow delays in communication, with the result that messages can be delivered out-of-order, then the message passing technique described in this chapter breaks down and it becomes necessary to introduce ordering tags on messages. Such considerations mean that model checking can only complement, and cannot replace conventional system testing. Nonetheless, if one is aware of these limitations, then model checking can be applied to good effect, as it has in this chapter, to the verification of complex concurrent systems.

Chapter 11

Conclusions and Further Work

The ML family of languages has been both the source and the subject of much recent research in the area of types in compilation. This area concerns the cross-fertilisation of the theory of type systems with the practical concerns of compiler writers. A great deal of discussion has taken place in the functional programming community over the advantages and disadvantages of the retention of type information throughout the compilation process, and then into the runtime system. Nonetheless, there appears to be an emerging consensus that the advantages afforded by type-based approaches outweigh any performance penalties incurred through the maintenance of type information. Much of the research and discussion has been directed at improving existing compilation technology. However, in this thesis we advance the argument with the claim that types in compilation also permit a number of language extensions that would otherwise be difficult or impossible to express. Type information is used throughout this thesis for a wide variety of purposes. However, the three novel extensions for which we have used type information are summarised below:

1. An efficient tag-free garbage collection operation.

2. An effective platform for distributed evaluation and garbage collection.

3. A powerful model of sequential and distributed code-replacement.

Other researches are also beginning to explore the potential of type information in other application areas, for example, type-based security models [LR98], type-directed partial evaluation [BD98], and a typed assembly language generation [MCG$^+$99]. In the remainder of this final chapter we will summarise the contributions and results of the thesis in detail, concluding with an outline of a number of possible extensions and areas for further research.

11.1 Thesis Summary

In this thesis we have defined a new language called Dynamic ML, a variant of the Standard ML language with extensions for code-replacement. The visible changes to the Standard ML language are minimal, while the underlying compilation and evaluation frameworks have been radically altered. The foundation for all of the definitions in the thesis is our $\mathcal{M}\Lambda$ language, presented in Chapter 2. $\mathcal{M}\Lambda$ is a clean, typed, λ-calculus-based *intermediate* language, similar in style to a number of other intermediate languages used in the compilation of Standard ML. The $\mathcal{M}\Lambda$ language is a good compromise between the theoretical simplicity of System F_2, and the practical complexity of Dynamic ML. The main advantage of using such an intermediate language is that it allows a style of definition which is closely related to the complete Dynamic ML language, while avoiding a great deal of unnecessary complexity. Furthermore, the $\mathcal{M}\Lambda$ language avoids the over-simplicity of System F_2, where it is difficult to relate to the dominant issues of a real implementation.

We have made continual reference to types and type systems throughout the thesis. Indeed, the use of types, and the type system of $\mathcal{M}\Lambda$ is central to the correctness of our definitions. Thus, in Chapter 3 we have defined the $\mathcal{M}\Lambda$ type system, called the static semantics, in intricate detail. In this chapter we also illustrated the relationship between System F_2 and $\mathcal{M}\Lambda$ through a comparison of their type systems.

In order to facilitate an implementation of the code-replacement technique, and to firmly establish the connection between Dynamic ML and $\mathcal{M}\Lambda$, we have defined a detailed translation between the core of these languages in Chapter 4. A central part of this translation is the conversion of the implicit types of Dynamic ML into explicit $\mathcal{M}\Lambda$ types through the process of type inference. Type inference in Standard ML is performed using Milner's Algorithm $\mathcal{W}$ [Mil78]. However, while variants of this algorithm are used in all of the current Standard ML compilers, there are no actual papers which describe all of the necessary details for the full language. Therefore, this chapter will also serve as a useful reference for the Standard ML compiler writer.

In Chapter 5 we observed that the dynamic semantics of Standard ML, as presented in the language definition [MTHM97], is at too high a level of abstraction for code-replacement. In particular, all of the details of memory management are omitted. Consequently, we have defined an alternative dynamic semantics in this chapter which exposes the crucial details of the memory, the heap and the stack, without placing any unnecessary restrictions on the layout or addressing of the memory. The dynamic semantics were presented through a novel type-preserving abstract machine. The advantages of this approach are twofold: our abstract machine definition provides both a sound theoretical basis for our garbage collection and code-replacement operations, and a practical basis for an implementation in the same way that the Java Virtual Machine (JVM) provides a platform for the Java language. In order to address concerns regarding the performance overheads of maintaining runtime type information, we also presented a novel *deferred type-instantiation* scheme for efficiently representing the polymorphic types.

Modern programming languages often perform a garbage collection operation on the memory to discard any redundant data. In Chapter 6 we illustrated how the presence of runtime types enables an efficient *tag-free* style of collection. We presented an abstract machine description of the well known two-space copying garbage collection algorithm for the $\mathcal{M}\Lambda$ language. This abstract machine operates in tandem with our earlier evaluation abstract machine, periodically freeing-up memory when required.

The Standard ML programming language contains a very powerful modules system, designed to support the construction of large programs. An adaptation of this modules system to facilitate code-replacement in Dynamic ML was presented in Chapter 7. The main alterations being the removal of transparent signatures, which are incompatible with code-replacement, and the addition of a `where datatype` construct to lessen the restrictions imposed by opaque signatures. We presented an innovative translation from the Dynamic ML modules language into $\mathcal{M}\Lambda$. The main feature of this system being that no extra code is generated for the module, rather the modules system acts only as a compile-time linking language. There are two kinds of replacement which we admit in Dynamic ML. The first kind is the replacement of one module with another, provided that both modules are matched by the same signature. The second kind of replacement permits the substitution of a signature with a more permissive one. The combination of these enables a very flexible, yet completely type-safe, replacement operation. In the second-half of the chapter we presented our code-replacement algorithm which performs the runtime replacement of types and values as an extension of the tag-free garbage collection technique.

In Chapter 8 we move from a sequential view of evaluation to a distributed configuration. In the sequential case, our assumption was that operations occurred in sequence, on a single processor of a single machine. In the distributed model we relax these constraints and permit processes containing multiple threads evaluating on a single machine, and multiple processes evaluating on multiple machines. This model can represent a range of different architectures, including symmetrical multiprocessing (SMP) machines, networks of workstations, and multiprocessing parallel (MPP) computers. The underlying assumption in our distributed model is that the interconnecting network is reasonably reliable and responsive. The $\mathcal{M}\Lambda$ language defined in Chapter 2 contains operations for creating threads of execution both locally and remotely, and for communicating between threads across bi-directional typed channels. In this chapter we build upon our earlier work on the LEMMA shared-memory interface, a platform for distributed evaluation in the Poly/ML compiler. We provide a definition of the LEMMA interface as a distributed abstract machine for $\mathcal{M}\Lambda$. The key innovation in this chapter is the use of *traces*, which allow a big-step style of semantics as opposed to the usual small-step models for expressing concurrency. In order to improve the efficiency of an implementation, we outline a technique for the caching of immutable data in the distributed system. This is an optimisation which is not possible for non-functional languages. Once again, the runtime type information plays a crucial role in determining the immutable data items.

We turn our attention to providing distributed definitions of garbage collection and code replacement in Chapter 9. Both of these algorithms can be viewed as generalisations of the sequential case into a distributed setting. This was not by accident, since one of our main reasons for selecting the two-space collection algorithm was the existence of a natural distributed extension. The distributed garbage collection algorithm defined using an abstract machine, similar to the distributed evaluation. As before, distributed code-replacement is a relatively straightforward extension of garbage collection. No changes to the Dynamic ML module system are necessary when moving to a distributed setting.

The issues of verification and validation of our definitions are addressed in Chapter 10. This is particularly important as we do not currently have a complete implementation. We chose to focus the verification on the distributed garbage collection algorithm, since this is central to the code-replacement technique. Furthermore, the sequential algorithm is captured as a special case of the distributed algorithm. Owing to the complexity of the definition, we have adopted an automated approach to the verification process, using the SPIN model checker. In this chapter we reformulated the garbage collection algorithm as a state-transition model in the PROMELA language. A significant number of correctness properties were subsequently verified against the model, and no significant problems were uncovered.

11.2 Thesis Achievements

In the conclusion of Chapter 1 the main aims of the thesis were outlined. We will now briefly discuss the extent to which these aims have been achieved and the limitations of the work as presented.

The main aim of the thesis was the definition of a powerful code replacement strategy for Standard ML. The intention was to surpass existing code-replacement facilities with the ability to update program data in addition to program code. It was also considered highly desirable to provide a guarantee of correctness of replacement. We believe that this aim has been largely accomplished, and we consider this accomplishment to be the main contribution of this thesis. Our code replacement strategy utilises an unorthodox technique which performs code replacement in conjunction with a type-based garbage collection operation. This is based on the observation that garbage collection already interacts with all of the program data relevant to replacement. By founding our technique on strong static typing we achieve the necessary correctness guarantees.

The second aim of the thesis was a detailed definition for all of the stages of code replacement, from the source language (Dynamic ML) through to the point at which replacement is actually performed. This was intended to mirror the stages of a real compiler for the language. We believe that this aim has also been accomplished and that there is significant novelty in this work. This includes the abstract machine semantics, the modules translation technique, and the tag-free garbage collection operation.

The final aim that we outlined involved the definition and validation of a distributed extension of our code replacement technique. The intention here being a demonstration of the applicability of our work to future computing demands. We believe that this aim has also been largely met by the provision of an abstract-machine semantics for the LEMMA interface, a distributed implementation of our algorithm, and the subsequent model-checking verification.

The limitations of our work currently lie in the areas of validation and implementation. Although the model checking verification goes some way towards establishing the correctness of our algorithms, it would nonetheless be desirable to apply a range of other validation techniques, such as formal proof (e.g. type soundness) and traditional testing strategies. However, these are largely dependent on the provision of an implementation, which has not yet been performed. The issues associated with implementation are detailed in the following section. It should be noted that these limitations do not undermine the accomplishments of the thesis and it is intended that they will be rectified as future work.

11.3 Implementation Issues

Throughout this thesis, the use of type-information has been our guideline. As we have seen, this has allowed us to produce clean, safe, and effective definitions. In some cases, these definitions have been a reformulation of existing algorithms, and in other cases, type information has enabled a completely novel approach. However, pragmatic considerations have also played a very important role in our definitions. We will now turn our attention to the issues associated with the provision of an implementation of Dynamic ML. Although we have yet to implement Dynamic ML, we nonetheless have a high degree of confidence that a practical and efficient implementation is possible. This is primarily a result of our restriction to operational models throughout, in particular our heavy reliance on abstract machine formalisms.

The ML-Kit compiler [TBE+98] is a direct implementation of the semantics of the Definition of Standard ML [MTHM97]. Each semantic rule in the definition has a function counterpart in the compiler. This compiler effectively demonstrates the practicality of a direct implementation based on the semantics of the language. The advantage of this approach is that it ensures that the language definition is faithfully reproduced in the implementation. It is our intention that the semantics of Dynamic ML should also be directly implemented in this manner. The translation of the semantics in the ML-Kit compiler was done manually. It is worth noting that there has been some recent research into automating the translation from the semantics, e.g. [Pet99], in a similar manner to the way in which tools such as Lex and Yacc currently automate the translation of the syntax. However, this work has not yet reached a level where we can use it to automate the translation of the Dynamic ML semantics.

In the introductory chapter we discussed the advantages of the abstract machine style of dynamic semantics over the natural-semantics style used in the Definition of Standard ML. It is interesting to note that the initial revision of the ML-Kit compiler contained a direct interpreter for the dynamic semantics. However, later revisions have performed a translation into an abstract machine semantics, and have experienced a considerable improvement in performance as a result. We have stressed the advantages of the abstract machine approach throughout the thesis, namely that is establishes the correct level of detail for our models, and that it enables an efficient implementation owing to the finite deterministic behaviour of abstract machines. Indeed, abstract machine models have long been at the foundation of implementations of functional languages, though they have not traditionally been associated with high performance.

Research into the efficient implementation of abstract machine models has recently been rekindled by the Java language. Java uses a form of abstract machine (the JVM) to enable portability of programs across architectures without re-compilation. Owing to the popularity of the Java platform, much effort has been devoted to optimising the performance of the JVM. The favoured approach is a just-in-time translation between the virtual machine instructions, called *byte-codes*, into native machine code at runtime. The current state-of-the-art is the Hotspot virtual machine [Gri98] which uses a range of analyses to ensure that the overheads of the translation do not outweigh the performance advantages. Benchmarks have shown that the performance of the Hotspot virtual machine can approach that of native C under certain conditions.

The effectiveness of such runtime code generation techniques when applied to functional languages has been demonstrated in the construction of the Fabius compiler for ML [LL96]. Using this technology it is possible for us to eliminate the runtime penalties incurred by the use of abstract types in module specifications by exploiting the underlying representation of an abstract type and re-compiling at runtime when the replacement module is available. Further, many other benefits come from the use of runtime code generation including those associated with *partial evaluation* [JGS93], since it is possible to take advantage of values which are not known until runtime.

The majority of ML compilers are constructed in some variant of the ML language itself. It is our intention to implement the Dynamic ML language using existing Standard ML compilers. This follows the accepted compiler *boot-strapping* tradition. It may be surprising that Standard ML is suitable for compiler construction given the very high-level nature of the language. However, the richness of the language makes a very clean and straightforward implementation possible. To illustrate this point we have defined the syntax of the $\mathcal{M}\Lambda$ language from Chapter 2 as a signature in Figure 11.1, and the syntax of the sequential abstract machine from Chapter 5 in Figure 11.2. Note the very clear resemblance between the abstract syntax and the datatype definitions. It will not be possible to define Dynamic ML in its entirety using only Standard ML, since we require a number of lower-level facilities that are not available in the language. These include direct memory access for garbage collection and code-replacement, and inter-process communication for distributed execution. Therefore, in common with other ML compilers, we must augment the Standard ML definition with a *runtime system* defined in a language such as C. This will likely be only a minimal set of C functions available through a foreign-language interface facility.

```
signature MLAMBDA =
  sig
    type 'a seq
    type 'a set
    type ('a, 'b) fmap
    type con
    type alpha
    type x
    type tn

    datatype scon = Unit | Int of int | Word of word | Real of real
                 | Char of char | String of string
    datatype tau = Tname of tn
                 | Tcons of tn * (tau seq)
                 | Ttuple of tau seq
                 | Tfunc of tau * tau
                 | Tvar of alpha
    datatype sigma = Mono of tau | Poly of (alpha seq) * tau
    datatype P = Program of (D set) * (X set) * E
    and D = Datatype1 of tn * C set
          | Datatype2 of alpha seq * tn * C set
    and X = Exception of C set
    and C = Nullary of con | Unary of con * tau
    and E = Scon of scon
        | Var1 of x
        | Var2 of x * tau seq
        | Fn1 of x * tau * tau * E
        | Fn2 of x seq * tau seq * tau * E
        | Con1 of con
        | Con2 of con * E
        | Con3 of con * tau seq
        | Con4 of con * tau seq * E
        | Decon1 of con * E
        | Decon2 of con * (tau seq) * E
        | Tuple of E seq
        | Select of int * E
        | Let of x * sigma * E * E
        | Let2 of (x * sigma) seq * E * E
        | Switch of E * ((con, E) fmap) * E
        | Fix of (x * sigma * E) seq * E
        | App1 of E * E
        | App2 of E * (E seq)
        | Assign of E * E
        | Raise of E * tau
        | Handle of E * E
  end;
```

Figure 11.1: $\mathcal{M}\Lambda$ *Represented in Standard ML.*

```
signature MACHINE =
  sig
    type 'a seq
    type ('a, 'b) fmap
    type l
    type p
    type tn
    type alpha
    type scon
    type con
    type x
    type E
    type P

    datatype M = Machine of H * Delta * XS * RS
    and H = Heap of TH * VH
    and TH = THeap of (p, ty) fmap
    and ty = RTname of tn
           | RTcons of tn * p
           | RTtuple of p seq
           | RTfunc of p * p
           | RTvar of alpha
           | RTclosure of TE * p
    and VH = VHeap of (l, va) fmap
    and va = RVscon of scon
           | RVncon of con
           | RVucon of con * l
           | RVtuple of l seq
           | RVclosure of Delta * x seq * E
           | RVomega
    and Delta = Env of TE * VE * DE * CE
    and TE = TEnv of (alpha, p) fmap
    and VE = VEnv of (x, (l * p)) fmap
    and DE = DEnv of (tn, p) fmap
    and CE = CEnv of con * p
    and XS = XSEmpty
           | XSva of (l * p) * XS
    and RS = RSEmpty
           | RSty of p * RS
           | RSva of (l * p) * RS
           | RSenv of Delta * RS

    val lpointer : int ref
    val ppointer : int ref
    val eval : M * P -> M
  end;
```

Figure 11.2: *Abstract Machine Description in Standard ML.*

11.4 Performance Modelling

One of the motivating factors behind the code-replacement technique is that improvements to the performance of programs can be introduced. However, without assistance, it is often difficult to establish where there is a performance bottleneck in a program, or what the impact of an improvement will be. The solution to these problems lies in the area of *performance modelling* techniques. The aim of these techniques is to establish the time and space requirements for a given fragment of code. Performance modelling techniques are generally divided into *static* and *dynamic* analyses. Static analysis is an attempt to determine the behaviour based purely on the source code of the program, i.e. at compile time, while a dynamic analysis determines the behaviour during evaluation, i.e. at runtime. We will only discuss dynamic techniques here, as static techniques tend to be very complex and specialised, owing to the fact that execution time of a program in the general case is undecidable. Furthermore, dynamic techniques appear to be a better fit for code-replacement, which is itself a runtime operation.

Dynamic performance modelling is usually achieved by *profiling* techniques; information is recorded during the evaluation of the program for subsequent analysis. There are two principal approaches to the collection of profiling information. In a *sample-based* approach, the program is interrupted at regular intervals, and the position of the program counter is recorded with reference to a timer. By using a number of different timers, execution time can be attributed to different parts of a program. A profiler for Standard ML using the sample-based approach, with a different timer for each function, is defined in [ADM88]. The alternative is a *counter-based* approach, where frequency counters are placed directly inside the program. This can be done implicitly, i.e. a counter is automatically placed at the beginning of each function, or explicitly, i.e. in a manner similar to setting breakpoints in debugging. The frequency counters are incremented by the path of execution. A counter-based profiler for the Lazy ML language is outlined in [RW93].

A novel counter-based profiling technique, which appears particularly relevant to Dynamic ML, is outlined in [Bra94] and later refined in [SP97]. In this strategy, profiling costs are related to the dynamic semantics of the language, rather than to a particular implementation. While this may not always yield the best results, when the implementation diverges considerably from the semantics, we intend an implementation of Dynamic ML to closely follow our abstract machine definitions. The technique is most easily illustrated with the aid of an example. Consider the rule from Chapter 5 for the application of a function, shown in Figure 11.3. The cost of a function application can be seen to comprise four separate costs: the cost of evaluating the function expression E_1, the cost of evaluating the argument expression E_2, the cost of dereferencing the heap $H_3(l'_1)$ to obtain the closure, and the cost of evaluating the closure expression E_c.

$$\frac{\begin{array}{l}(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ E_1) \Rightarrow (H_2,\ \Delta_2,\ XS_2,\ (l_1,\ p_1)\cdot RS'_1)\\ (H_2,\ \Delta_2,\ XS_2,\ (l_1,\ p_1)\cdot RS'_1,\ E_2) \Rightarrow (H_3,\ \Delta_3,\ XS_3,\ (l_2,\ p_2)\cdot(l'_1,\ p'_1)\cdot RS''_1)\\ H_3(l'_1) = \langle\!\langle \Delta_c,\ (x),\ E_c \rangle\!\rangle\\ (H_3,\ \Delta_c[x \mapsto (l_2,\ p_2)],\ XS_3,\ \Delta_3\cdot RS''_1,\ E_c) \Rightarrow\\ \quad (H_4,\ \Delta_4,\ XS_4,\ (l_3,\ p_3)\cdot\Delta'_3\cdot RS'''_1)\end{array}}{(H_1,\ \Delta_1,\ XS_1,\ RS_1,\ \mathbf{app}\ (E_1,\ E_2)) \Rightarrow (H_4,\ \Delta'_3,\ XS_4,\ (l_3,\ p_3)\cdot RS'''_1)}$$

Figure 11.3: *Function Application*

A simple measure of the performance for the entire program can be obtained by counting the number of auxiliary rules invoked at each step of the evaluation. A measure of the performance for each function can also be obtained by maintaining a separate count for every function expression. However, this is really just a *coverage* analysis, indicating which parts of the program are evaluated with the highest frequency. A measure of the time taken for the program can be obtained by scaling the result with a time constant, but this will bear little relation to the cost of a real implementation of the code.

A more subtle approach is needed to obtain a time value that is closer to the actual cost of an implementation. The problem arises from the fact that simple step-counting ignores a number of costs, such as the dereferencing of the heap in the above example. Also, step-counting does not take into account the fact that different operations will take different times to complete. A significant improvement can be made by associating a weight with each rule. Basis operations, such as the arithmetic and boolean operations, will also have associated weights. Ideally, these weights are obtained from actual empirical measurements of an implementation of each rule. This model will yield a much closer reflection of the cost of a real implementation. Consequently, this will enable performance bottlenecks to be easily identified, and improvements to the code can be compared with a reasonable degree of accuracy.

Although the use of weights will yield a reasonable estimate of performance in many cases, the results will be distorted by any optimisations which are performed on the code. For example, tail-calls are often optimised using loops. Hence, the cost of the top-level invocation of a function will be different from the cost of a recursive call. Furthermore, techniques such as just-in-time compilation will result in a significant skewing on any time measurements. These problems clearly need some careful consideration. However, it is likely that they can be overcome, e.g. by using multiple weightings and averages. Although it may not be possible to obtain entirely accurate time prediction in every case using the semantic-based approach, the technique does appear to hold considerable promise, particularly in the identification of hot-spots and redundant areas of programs.

The profiling techniques that we have discussed so far have been designed to analyse the *time* requirements of a program. However, there are also many profiling techniques whose purpose is to analyse *space* requirements. At the most basic level, this simply amounts to determining the heap usage for a program, or the individual functions within a program. As with time profiling, both sample-based and counter-based approaches can be used. Clearly, there will be some interaction between space profiling and garbage collection. Indeed, one of the main goals of space profiling is to minimise the space usage of a program, and therefore reduce the number of garbage collections that are required.

From a code-replacement perspective, a more interesting form of space profiling is to determine the frequency with which certain data structures are used, i.e. data profiling. For example, if a certain queue data structure is used repeatedly in a program, then it would make a good candidate for an improved implementation, e.g. using a fixed array instead of a list. The profiling of data structures in Dynamic ML can also be achieved using the dynamic semantics of the language, since all data accesses are visible as references using pairs $(l,\ p)$. The presence of runtime type information makes this approach ideal, where in other languages explicit tags must be used. Given the type information associated with each data item, it is trivial to record the frequency of occurrence of certain data structures within an executing program.

One final area of interest concerning the interaction between code replacement and profiling is the appealing prospect of semi-automating the replacement operation. With a little effort, the profiling of a Dynamic ML program could be arranged to automatically identify candidate functions and data structures for optimisation, without requiring user interaction. A code optimisation process can then examine these candidates and perform a number of safe optimisations on the program, which will be introduced through garbage collection and code-replacement, as illustrated in Figure 11.4. A more sophisticated approach may introduce a variety of small changes into a running program and observe their effects by profiling, similar to genetic or evolutionary programming techniques.

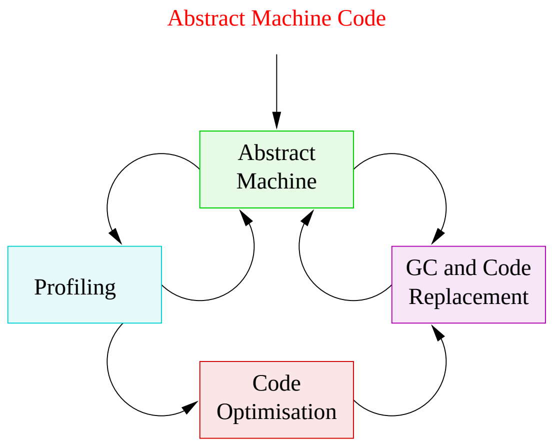


Figure 11.4: *Automated Code-Replacement.*

The design of an automated code optimisation process is clearly non-trivial and would require a significant research effort. Nonetheless, there are a number of simpler optimisations which can be applied at runtime and may result in a substantial increase in performance. We can improve the performance of frequently accessed data structures and functions by a technique called *cache-conscious data placement* [CHL99]. The objective of this optimisation is to improve data-locality and thereby reduce the cache-miss rate. There are a number of methods for improving locality, such as grouping functions together with their data, or grouping frequently accessed and related data items together. More aggressive optimisation can be performed with a knowledge of the actual cache size and structure of the executing platform. A number of researchers have experimented with a variety of layouts and have reported runtime improvements as much as 24% [Kis99]. This technique can be applied as a garbage collection operation, or by rewriting the code to improve the locality behaviour. It is likely that runtime knowledge of the type of the data could be used to improve the technique further. In a distributed setting, a locality analysis can be performed on communication between machines, e.g. [Kı99, Mor00]. Threads which communicate frequently can potentially be migrated together to improve performance. Finally, although we cannot arbitrarily rewrite functions at runtime, we can selectively improve their performance through a range of automatic parallelisation techniques, for example, using the implicit multi-threading operation described at the end of Chapter 8, or related techniques including parallel algorithmic skeletons [HM99]. It is clear that the area of automated profiling using explicit runtime type information is ripe for further research.

11.5 Code-Replacement in Java

In this thesis we have defined code-replacement exclusively in the context of the ML family of languages. Nonetheless, it has become apparent that the underlying techniques can be applied to a range of different programming platforms. In principle, our model of replacement may be applied to any statically typed language that is amenable to garbage-collection. It would be relatively straightforward to modify the technique for other functional languages, including lazily evaluated languages such as Haskell. However, we have recently begun to consider the application of code-replacement to the object-oriented language Java [GJS96]. Code-replacement in Java is intuitively appealing because useful, soundly-engineered products such as Java can be deployed in contexts where it is presently difficult to apply Standard ML. These would include embedded systems, wide-area mobile platforms such as the Internet, and also application domains where inter-operability is an important consideration.

As noted in the introductory chapter, the Java language already contains a restricted form of code-replacement called *binary-compatibility*. This allows the runtime replacement of class definitions which match certain (flawed) conditions [DEW99], though the instances of the class are not replaced. Another kind of code-replacement which enables Java objects to mutate, that is to change their class-membership, is defined in [DDDG99]. However, what we are proposing essentially combines both of these techniques. We wish to define a form of code-replacement, where one Java class is replaced with another, and all of the object instances of the class are replaced with the new representation. Owing to the object-oriented nature of classes, this will also require the replacement of any object instances of any *sub-classes*.

The kind of replacement which we are proposing is clearly a significantly more complex operation that the replacement of modules in Dynamic ML. We cannot resort to the specialisation of classes as we did for ML functors, and we must take into account the class hierarchy. Nonetheless, we believe that the underlying principles are similar enough to make such a replacement feasible. Indeed, this kind of replacement is the goal of the Java Distributed Run-time Update Management System JDRUMS [RA00]. However, the implementation of replacement in this system is somewhat different from ours. In JDRUMS, the replacement of objects of the outdated class is performed lazily as the objects are accessed, meaning that both versions of the class are active at the same time. Objects are garbage-collected and replaced as they are accessed during evaluation. When all of the objects of the old version of the class have been replaced, the class object will have no more references and it can then be removed by the garbage-collector. It is interesting to note that the JDRUMS system suffers from the same limitation as ours in that active code cannot be replaced. The lazy approach to replacement is in fact more suitable than the all-at-once replacement in Dynamic ML when dealing with the Java language, given that there are likely to be a large number of instances of each class. Moreover, this technique is also more suitable for wide-area implementations, given that communication delays tend to make a global garbage collection operation impractical. However, where the JDRUMS system falls short is that type information is not used to ensure the safety of the replacement. Consequently, it is this area that we wish to target with our own Java replacement technique.

There are a number of hurdles which must be overcome if we are to define a type-safe code-replacement operation in Java. These primarily concern typing aspects of the language. The Java language is both statically and dynamically typed. For example, method invocations and field accesses are statically typed, while array accesses and casts are dynamically typed. This in itself is not necessarily a problem, as we have shown in the introductory chapter that dynamic types can be safely incorporated into a statically typed language. The problem is that the type system of Java is only informally defined in [GJS96]. This type system is ambiguous in a number of areas, and indeed several aspects of the Java language have been shown to violate type-safety [Sar97]. In order to define a type-safe code-replacement operation, we require a formal type system that is known to be type safe.

The definition of a formal type system for the complete Java language is a somewhat daunting prospect, as the language contains many complex features including sophisticated concurrency, security, and dynamic-linking mechanisms. The complete type system must include aspects of both static and dynamic typing, sub-typing, and polymorphic types. Nonetheless, there have already been a number of successful attempts to formalise subsets of the Java language, two of which are Java$_{light}$ [NvO98] and Java$_{\mathcal{S}}$ [DEK99]. We have also defined our own formal subset of Java for use in an earlier prototype ML to Java translation system [Wal98], the syntax of which is shown in Figure 11.5. Our subset has been renamed DJ, an abbreviation for Dynamic Java, in this thesis. The subset covers the majority of basic Java definitions for classes, interfaces, methods, fields, and arrays. The meta-variables *cn*, *in*, *m*, *v*, *c*, and *gt* are used for class names, interface names, method names, variables, constants, and ground-types (`int`, `float`, `boolean`, etc.) respectively.

$ty \in$ Type	$::=$	`null`	(null type)
	$\mid$	gt	(ground type)
	$\mid$	cn	(class type)
	$\mid$	$cn[\,]$	(array type)
	$\mid$	in	(interface)
$mt \in$ MType	$::=$	$at \rightarrow ty$	(method type)
$at \in$ AType	$::=$	$\overline{ty}^k$	(argument type)
$PR \in$ Program	$::=$	$(\overline{\overline{CD}},\ \overline{\overline{IF}})$	
$CD \in$ Class	$::=$	**class** $(cn_1,\ cn_2,\ \overline{\overline{in}}) = (\overline{\overline{CX}},\ \overline{\overline{F}},\ \overline{\overline{M}})$	
$CX \in$ Constructor	$::=$	**constructor** $(\overline{v}^i,\ at) = (\overline{E}^j,\ \overline{S}^k)$	
$F \in$ Field	$::=$	**field** $(v,\ ty)$	
$M \in$ Method	$::=$	**method** $(m,\ \overline{v}^j,\ mt) = (\overline{S}^k,\ E)$	
$IF \in$ Interface	$::=$	**interface** $(in_1,\ in_2) = \overline{\overline{IM}}$	
$IM \in$ IMethod	$::=$	**method** $(m,\ \overline{v}^j,\ mt)$	
$S \in$ Statement	$::=$	E	(expression)
	$\mid$	**local** $(v,\ ty) = E$ **in** $\overline{S}^k$	(declaration)
	$\mid$	**if** E **then** $\overline{S_1}^j$ **else** $\overline{S_2}^k$	(conditional)
	$\mid$	**throw** E	(throw exception)
	$\mid$	**try** $\overline{S_1}^j$ **catch** $(cn,\ v,\ \overline{S_2}^k)$	(catch exception)
$E \in$ Expression	$::=$	V	(variable)
	$\mid$	c	(literal)
	$\mid$	**null**	(null object)
	$\mid$	**this**	(current object)
	$\mid$	$cn[\overline{E}^k]$	(array)
	$\mid$	$V := E$	(assignment)
	$\mid$	**apply** $(E_1,\ m,\ \overline{E_2}^k)$	(invocation)
	$\mid$	**instance** $(cn,\ \overline{E}^k)$	(class instance)
$V \in$ Variable	$::=$	v	(local variable)
	$\mid$	$E_1[E_2]$	(array variable)
	$\mid$	$E\ .\ v$	(field)

Figure 11.5: *Abstract Syntax of DJ*

The abstract syntax of DJ will not look particularly familiar to the Java programmer. This is because we believe that a great deal of the complexity of the Java language can be eliminated at compile-time, just as we did for Dynamic ML in Chapter 4. Hence, DJ is defined in the minimalistic style of an intermediate language, similar to the $\mathcal{M}\Lambda$ language. For example, a class declaration **class** $(cn_1,\ cn_2,\ \overline{\overline{in}}) = (CX,\ \overline{\overline{F}},\ \overline{\overline{M}})$, defines a new class named cn_1 which extends an existing class cn_2 and implements the set of (possibly empty) interfaces $\overline{\overline{in}}$. Every class contains a set of constructors $\overline{\overline{CX}}$, a set of fields $\overline{\overline{F}}$, and a set of methods $\overline{\overline{M}}$. The remaining language constructs should be reasonably self-explanatory. The DJ subset is designed to be entirely statically typed as we have removed such features as dynamic arrays and casts. This will further reduce the complexity of the code-replacement.

The other main issue that must be addressed is the definition of an abstract machine model for evaluating Java programs. The current definition of the JVM [LY96] is inadequate as it does not retain enough runtime typing information. It is interesting to note that alternative definitions to the JVM were also required in [NvO98] and [DEK99] for a proof of type soundness. We have taken the first steps toward the definition of an abstract machine for evaluating our DJ subset. The abstract syntax is presented in Figure 11.6.

Machine State	M	$::=$	$(H,\ \Delta,\ XS,\ RS)$	
Heap	H	$::=$	$(TH,\ VH)$	
Pointer	p			
Type Heap	TH	$::=$	$p \overset{map}{\mapsto} ty$	
Heap Types	ty	$::=$	gt	(ground type)
		$\mid$	cn	(class type)
		$\mid$	$p[\,]$	(array type)
		$\mid$	$\overline{p}^{k}$	(argument type)
		$\mid$	$\overline{p_1}^{k} \to p_2$	(method type)
		$\mid$	`null`	(null type)
Location	l			
Value Heap	VH	$::=$	$l \overset{map}{\mapsto} obj$	
Heap Objects	obj	$::=$	c	(constant)
		$\mid$	$[\overline{l}^{k}]$	(array)
		$\mid$	$\langle v \overset{map}{\mapsto} l,\ m \overset{map}{\mapsto} (l,\ p)\rangle$	(class object)
		$\mid$	$\langle\!\langle \overline{v}^{j},\ \overline{S}^{k} \rangle\!\rangle$	(method closure)
		$\mid$	$null$	(null value)
Environment	Δ	$::=$	$(TE,\ CE,\ IE,\ VE)$	
This	TE	$::=$	$this \mapsto (l,\ p)$	
Class Env.	CE	$::=$	$cn \overset{map}{\mapsto} (cn',\ \overline{\overline{in}},\ FE,\ ME)$	
Interface Env.	IE	$::=$	$in \overset{map}{\mapsto} (in',\ ME)$	
Field Env.	FE	$::=$	$v \overset{map}{\mapsto} p$	
Method Env.	ME	$::=$	$m \overset{map}{\mapsto} (l,\ p)$	
Variable Env.	VE	$::=$	$v \overset{map}{\mapsto} (l,\ p)$	
Exception Stack	XS	$::=$	$() \mid (l,\ p) \cdot XS$	
Result Stack	RS	$::=$	$() \mid (l,\ p) \cdot RS$	

Figure 11.6: *An Abstract Machine Model for DJ.*

The top-level organisation of the machine is identical to the $\mathcal{M}\Lambda$ abstract machine, but the contents of the heap and the environment have been adapted for the Java language. In principle, this should make it easier to adapt our garbage collection and code-replacement operations to DJ. As before, the heap types correspond directly to types in DJ, and the heap objects belong to the heap types. We also use p for type pointers and v for value locations. The heap types ty are ground types gt, class instance types cn, array types $p[\,]$, method types $\overline{p_1}^k \rightarrow p_2$, and the null type `null`. The corresponding heap objects obj are constants c, class objects $\langle v \overset{map}{\mapsto} l,\ m \overset{map}{\mapsto} (l,\ p)\rangle$, array objects $[\bar{l}^k]$, method closures $\langle\!\langle \overline{v}^j,\ \overline{S}^k \rangle\!\rangle$, and the *null* object. The environment Δ maps the field, method, and variable identifiers to their corresponding objects on the heap. When evaluating inside a class body, *this* is mapped to the current class object. The class hierarchy is described by the class environment CE which maps each class name cn to its superclass cn'. The top of the hierarchy is assumed to be a special class named `Object` which has itself as superclass. The interface hierarchies are defined in a similar manner by the interface environment IE. The top of each interface hierarchy will be an interface which extends itself, since there is no equivalent of `Object` for interfaces.

There still remain a significant number of challenges to be solved before we can provide a type-safe model of code-replacement in Java. However, we believe that the definition of the DJ intermediate language shown here will significantly reduce the complexity of the task. We also believe that further research into the use of type information in Java will result in significant improvements in the performance and security of the Java platform, as they have done for the Standard ML language.

Bibliography

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic Typing in a Statically Typed Language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[ACPR95] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic Typing in Polymorphic Languages. *Journal of Functional Programming*, 5(1):111–130, January 1995.

[ADM88] Andrew W. Appel, Bruce Duba, and David B. MacQueen. Profiling in the Presence of Optimisation and Garbage Collection. Technical Report CS-TR-197-88, Princeton University, Department of Computer Science, Princeton, New Jersey, November 1988.

[App87] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, June 1987.

[App88] Andrew W. Appel. Re-opening Closures. Unpublished manuscript available at `http://www.cs.princeton.edu/~appel/papers/`, January 1988.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[App94] Andrew W. Appel. Hot-sliding in ML. Unpublished manuscript available at `http://www.cs.princeton.edu/~appel/papers/`, December 1994.

[AR98] Saleh E. Abdullahi and Graem A. Ringwood. Garbage Collecting the Internet: A Survey of Distributed Garbage Collection. *ACM Computing Surveys*, 30(3):330–373, September 1998.

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG*. Prentice Hall, second edition, 1996.

[BD98] Vincent Balat and Olivier Danvy. Strong Normalisation by Type-Directed Partial Evaluation and Run-Time Code Generation. In *Proceedings of the 1998 Workshop on Types in Compilation (TIC'98)*, number 1473 in Lecture Notes in Computer Science, pages 240–252, Kyoto, Japan, March 1998. Springer-Verlag.

[Ber91] Dave Berry. *Generating Program Animators from Programming Language Semantics*. PhD thesis, LFCS, Division of Informatics, University of Edinburgh, June 1991. Thesis Number CST-79-91.

[BKR98] Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java Byte-codes. In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 129–140, Baltimore, Maryland, September 1998. ACM Press.

[Bra94] Tore A. Bratvold. *Skeleton-Based Parallelisation of Functional Programs*. PhD thesis, Department of Computing and Electrical Engineering, Herriot-Watt University, November 1994.

[Bru97] Glenn Bruns. *Distributed Systems Analysis with CCS*. Prentice Hall Europe, 1997.

[Car83] Luca Cardelli. The Functional Abstract Machine (FAM). Technical Report TR-107, AT&T Bell Laboratories, May 1983.

[Car86] Luca Cardelli. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, number 242 in Lecture Notes in Computer Science. Springer-Verlag, 1986.

[Car87] Luca Cardelli. Basic Polymorphic Type-checking. *Science of Computer Programming*, 8(2):147–172, 1987.

[Car88] Luca Cardelli. Phase Distinctions in Type Theory. Unpublished manuscript available at `http://www.luca.demon.co.uk`, January 1988.

[Car97] Luca Cardelli. *Type Systems*, chapter 140, pages 2208–2236. The Computer Science and Engineering Handbook. CRC Press, 1997.

[CGP99] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[CHL99] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious Structure Layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, pages 1–12, Atlanta, Georgia, May 1999. ACM Press.

[DDDG99] Sophia Drossopoulou, Mariangiola Dezani-Ciancaglini, Ferruccio Damiani, and Paola Giannini. Objects Dynamically Changing Class. Work in progress - manuscript available at `http://www.di.unito.it/~dezani/papers/odcc.html`, August 1999.

[DEK99] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. *Is the Java Type System Sound?*, volume 5 of *Theory and Practice of Object Systems*, chapter 1, pages 3–24. John Wiley and Son, 1999. A later revision of this paper is available at `http://www-dse.doc.ic.ac.uk/projects/slurp/pubs.html`.

[DEW99] Sophia Drossopoulou, Susan Eisenbach, and David Wragg. A Fragment Calculus - towards a model of Separate Compilation, Linking and Binary Compatibility. In *Proceedings of the 1999 IEEE Symposium on Logic in Computer Science (LICS'99)*, Trento, Italy, April 1999.

[DF95] Jürgen Dingel and Thomas Filkorn. Model Checking for Infinite State Systems using Data Abstraction, Assumption-commitment style Reasoning and Theorem Proving. In *Proceedings of the Seventh International Conference on Computer Aided Verification (CAV'95)*, number 939 in Lecture Notes in Computer Science, pages 54–69, Liége, Belgium, July 1995. Springer-Verlag.

[Die96] Stephan Diehl. *Semantics-Directed Generation of Compilers and Abstract Machines*. PhD thesis, Fachbereich Informatik, Universität des Saarlandes, 1996.

[Dor98] Christopher Dornan. *Type-Secure Meta-Programming*. PhD thesis, Department of Computer Science, University of Bristol, July 1998.

[Dug96] Dominic Duggan. A Type-Based Implementation of a Language with Distributed Scope. In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, number 1222 in Lecture Notes in Computer Science, pages 277–293. Springer-Verlag, 1996.

[Els98] Martin Elsman. Polymorphic Equality - No Tags Required. In *Proceedings of the 1998 Workshop on Types in Compilation (TIC'98)*, number 1473 in Lecture Notes in Computer Science, pages 136–155, Kyoto, Japan, March 1998. Springer-Verlag.

[Els99] Martin Elsman. *Program Modules, Separate Compilation, and Intermodule Optimisation*. PhD thesis, Department of Computer Science, University of Copenhagen, January 1999.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java^{TM} Language Specification*. Addison-Wesley, 1996.

[Gri98] David Griswold. The Java Hotspot Virtual Machine Architecture. SUN Microsystems White Paper, March 1998.

[Ham91] Kevin Hammond. *Parallel SML: a Functional Language and its Implementation in Dactl*. Research Monographs in Parallel and Distributed Computing. Pitman Publishing, 1991.

[Hav99] Klaus Havelund. Mechanical Verification of a Garbage Collector. In *Proceedings of the 1999 International Parallel and Distributed Processing Symposium (IPDPS '99)*, Workshop on Formal Methods for Parallel Programming: Theory and Applications, San Juan, Puerto Rico, April 1999.

[Hen80] P. Henderson. *Functional Programming: Application and Implementation*. Prentice-Hall International, 1980.

[HM99] Kevin Hammond and Greg Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, November 1999.

[Hol97a] Gerard J. Holzmann. State Compression in SPIN: Recursive Indexing and Compression Training Runs. In *Proceedings of the 3rd SPIN Workshop (SPIN'97)*, Twente University, Enschede, The Netherlands, April 1997.

[Hol97b] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):1–17, May 1997.

[Hol98] Gerard J. Holzmann. An Analysis of Bit-state Hashing. *Formal Methods in System Design*, 13(3):289–307, November 1998.

[HS97] Robert Harper and Chris Stone. An Interpretation of Standard ML in Type Theory. Technical Report CMU-CS-97-147, School of Computer Science, Carnegie Mellon University, June 1997. Also published as Fox Memorandum CMU-CS-FOX-97-01.

[Jac98] Paul B. Jackson. Verifying a Garbage Collection Algorithm. In *Proceedings of 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'98)*, number 1479 in Lecture Notes in Computer Science, pages 225–244, Canberra, Australia, September 1998. Springer-Verlag.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, September 1993.

[JH93] Mark P. Jones and Paul Hudak. Implicit and explicit parallel programming in Haskell. Technical Report Research Report YALEU/DCS/RR-982, Yale University, August 1993.

[JL96] Richard Jones and Rafael D Lins. *Garbage Collection : Algorithms for Automatic Dynamic Memory Management*. John Wiley and Son, August 1996.

[Kah87] Gilles Kahn. Natural Semantics. In *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science (STACS'87)*, number 247 in Lecture Notes in Computer Science, pages 22–39, Passau, Germany, February 1987. Springer-Verlag.

[Kah93] Stefan Kahrs. Mistakes and Ambiguities in the Definition of Standard ML. Technical Report ECS-LFCS-93-257, LFCS, Division of Informatics, University of Edinburgh, April 1993.

[Kı99] Dilsun Kırlı. A polymorphic type and effect system for detecting mobile functions. Technical Report ECS-LFCS-99-413, LFCS, Division of Informatics, University of Edinburgh, October 1999.

[Kis99] Thomas Kistler. *Continuous Program Optimisation.* PhD thesis, Department of Information and Computer Science, University of California, November 1999.

[KMM97] Graham Kirby, Ron Morrison, and David Munro. Evolving Persistent Applications on Commercial Platforms. In *Proceedings of the First East-European Symposium on Advances in Databases and Information Systems (ADBIS'97)*, volume 1, pages 170–179, St. Petersburg, Russia, September 1997. Nevsky Dialect.

[Kra96] Douglas Kramer. *The Java Platform, A White Paper.* Sun Microsystems Inc., May 1996.

[Lan64] P. J. Landin. The Mechanical Evaluation of Expressions. *Computer Journal*, 6(4):308–320, 1964.

[Ler92] Xavier Leroy. *Polymorphic Typing of an Algorithmic Language.* PhD thesis, Institut National de Recherche en Informatique et Automatique (INRIA), 1992. Thesis No. 1778.

[Ler94] Xavier Leroy. Manifest Types, Modules, and Separate Compilation. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages (POPL'94)*, pages 109–122, Portland, Oregon, January 1994. ACM Press.

[Ler95] Xavier Leroy. *The CAML Special Light System, Documentation and User's Guide.* Institut National de Recherche en Informatique et Automatique (INRIA), release 1.12 edition, 1995. Available at `http://www.labomath.univ-orleans.fr/docs/csl/`.

[Ler98] Xavier Leroy. An Overview of Types in Compilation. In *Proceedings of the 1998 Workshop on Types in Compilation (TIC'98)*, number 1473 in Lecture Notes in Computer Science, pages 1–8, Kyoto, Japan, March 1998. Springer-Verlag.

[Lil97] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems.* PhD thesis, School of Computer Science, Carnegie Mellon University, May 1997. Thesis No. CMU-CS-97-122.

[LL96] Peter Lee and Mark Leone. Optimising ML with run-time code generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'96)*, pages 137–148, Philadelphia, Pennsylvania, May 1996. ACM Press.

[LM93] Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, October 1993.

[LR98] Xavier Leroy and François Rouaix. Security Properties of Typed Applets. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages (POPL'98)*, pages 391–403, San Diego, California, January 1998. ACM Press.

[LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine*. Addison-Wesley, September 1996.

[Mat91] David C.J. Matthews. A Distributed Concurrent Implementation of Standard ML. In *Proceedings of EurOpen Autumn 1991 Conference*, Budapest, Hungary, September 1991. Also published as LFCS Technical Report ECS-LFCS-91-17.

[MCG$^+$99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A Realistic Typed Assembly Language. In *Proceedings of the ACM SIGPLAN'99 Workshop on Compiler Support for System Software (WCSSS'99)*, pages 25–35, Atlanta, Georgia, May 1999. ACM Press.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[McM99] Kenneth L. McMillan. Verification of Infinite State Systems by Compositional Model Checking. Technical report, Cadence Berkeley Labs, 1999. `http://www-cad.eecs.berkeley.edu/~kenmcmil/papers/`.

[MH96] Greg Morrisett and Robert Harper. Semantics of Memory Management for Polymorphic Languages. Technical Report CMU-CS-96-176, School of Computer Science, Carnegie Mellon University, September 1996. Also published as Fox Memorandum CMU-CS-FOX-96-04.

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, April 1978.

[Mit94] Kevin Mitchell. Concurrency in a Natural Semantics. Technical Report ECS-LFCS-94-311, LFCS, Division of Informatics, University of Edinburgh, December 1994.

[MLP99] Andrew Moran, Sren B. Lassen, and Simon Peyton Jones. Imprecise Exceptions, Co-Inductively. In Andrew Gordon and Andrew Pitts, editors, *Proceedings of the Third International Workshop on Higher Order Operational Techniques in Semantics (HOOTS'99)*, volume 26 of *Electronic Notes in Theoretical Computer Science*, Paris, France, October 1999. Elsevier Science. Available at `http://www.elseview.nl/locate/entcs/`.

[MMH96] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed Closure Conversion. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 271–283, St. Petersburg Beach, Florida, January 1996. ACM Press.

[Mor00] Álvaro Moreira. *A Type-Based Locality Analysis for a Functional Distributed Language*. PhD thesis, LFCS, Division of Informatics, University of Edinburgh, April 2000. Thesis Number CST-156-00.

[MS95a] David C.J. Matthews and Thierry Le Sergent. LEMMA: A Distributed Shared Memory with Global and Local Garbage Collection. Technical Report ECS-LFCS-95-325, LFCS, Division of Informatics, University of Edinburgh, June 1995.

[MS95b] David C.J. Matthews and Thierry Le Sergent. LEMMA Interface Definition. Technical Report ECS-LFCS-95-316, LFCS, Division of Informatics, University of Edinburgh, January 1995.

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML*. The MIT Press, 1991.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML: Revised 1997*. The MIT Press, 1997.

[NL91] Bill Nitzberg and Virginia Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, pages 52–60, August 1991.

[NvO98] Tobias Nipkow and David von Oheimb. Java$_{light}$ is Type-Safe - Definitely. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages (POPL'98)*, pages 161–170, San Diego, California, January 1998. ACM Press.

[Obj98] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.2 edition, February 1998. Available at `http://www.omg.org/`.

[PDM89] Benjamin Pierce, Scott Dietzen, and Spiro Michaylov. Programming in Higher-Order Typed Lambda-Calculus. Technical Report CMU-CS-89-111, School of Computer Science, Carnegie Mellon University, March 1989.

[Pet99] Mikael Pettersson. *Compiling Natural Semantics*. Number 1549 in Lecture Notes in Computer Science. Springer-Verlag, 1999.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[Pey92] Simon L. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[RA00] Tobias Ritzau and Jesper Andersson. Dynamic Deployment of Java Applications. In *Proceedings of the 2000 Embedded Systems Show (ESS2000)*, London, England, May 2000.

[Red97] Frank E. Redmond, editor. *DCOM: Microsoft Distributed Component Object Model*. IDG Books Worldwide Inc., September 1997.

[Rey74] John C. Reynolds. Towards a Theory of Type Structure. In *Colloq. sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer-Verlag, 1974.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[RSS95] Sreeranga P. Rajan, Natarajan Shankar, and Mandayam K. Srivas. An Integration of Model Checking with Automated Proof Checking. In *Proceedings of the Seventh International Conference on Computer-Aided Verification (CAV '95)*, number 939 in Lecture Notes in Computer Science, pages 84–97, Liège, Belgium, July 1995. Springer-Verlag.

[Rus98] Claudio V. Russo. *Types for Modules*. PhD thesis, LFCS, Division of Informatics, University of Edinburgh, June 1998. Thesis Number CST-143-98.

[RW93] C. Runciman and D. Wakeling. Heap Profiling of Lazy Functional Programs. *Journal of Functional Programming*, 3(2):217–245, April 1993.

[Sar97] Vijay Saraswat. Java is not type-safe. Unpublished report available at `http://www.loria.fr/~lliquori/JAVA/bug.html`, August 1997.

[Ses96] Peter Sestoft. ML Pattern Match Compilation and Partial Evaluation. In *Seminar on Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 446–464. Springer-Verlag, 1996.

[SF93] Mark E. Segal and Ophir Frieder. On-the-Fly Program Modification: Systems for a Dynamic Updating. *IEEE Software*, 10(2):53–65, March 1993.

[Sha97] Zhong Shao. An Overview of the FLINT/ML Compiler. In *Proceedings of the 1997 Workshop on Types in Compilation (TIC'97)*, Amsterdam, The Netherlands, June 1997.

[SKA94] John Sargeant, Chris Kirkham, and Steve Anderson. The Uflow Computational Model and Intermediate Format. Technical Report UMCS Technical Report 94-5-1, Department of Computer Science, University of Manchester, 1994.

[SM94] Thierry Le Sergent and David C J Matthews. Adaptive selection of protocols for strict coherency in distributed shared memory. Technical Report ECS-LFCS-94-306, LFCS, Division of Informatics, University of Edinburgh, September 1994.

[SP97] Patrick M. Sansom and Simon L. Peyton Jones. Formally-based profiling for higher-order functional languages. *ACM Transactions on Programming Languages and Systems*, 19(1):334–385, January 1997.

[SSP98] Mark Shields, Tim Sheard, and Simon Peyton Jones. Dynamic Typing as Staged Type Inference. In *Proceeding of the 25th ACM Symposium on Principles of Programming Languages (POPL'98)*, pages 49–61, San Diego, California, January 1998. ACM Press.

[TBE$^+$98] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Olesen, Peter Sestoft, and Peter Bertelsen. Programming with Regions in the ML Kit (for Version 3). Technical Report DIKU-TR-98/25, Department of Computer Science, University of Copenhagen, December 1998.

[TMC$^+$96] David Tarditi, J. Gregory Morrisett, P. Cheng, C. Stone, Robert Harper, and Peter Lee. TIL: A Type-Directed Optimising Compiler for ML. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'96)*, pages 181–192, Philadephia, Pennsylvania, May 1996.

[Tof88] Mads Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, LFCS, Division of Informatics, University of Edinburgh, May 1988. Thesis Number CST-52-88.

[Tol94] Andrew Tolmach. Tag-free Garbage Collection using Explicit Type Parameters. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 1–11, Orlando, Florida, June 1994. ACM.

[TS97] Walid Taha and Tim Sheard. Multi-stage Programming with Explicit Annotations. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 203–217, Amsterdam, The Netherlands, June 1997.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-Based Memory Management. *Information and Computation*, 132(2):109–176, 1997.

[Var01] Moshe Y. Vardi. Branching vs. Linear Time: Final Showdown. In Tiziana Margaria and Wang Yi, editors, *Proceedings of the 2001 Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, number 2031 in Lecture Notes in Computer Science, pages 1–22, Genova, Italy, April 2001. Springer-Verlag.

[Wal98] Chris Walton. A Type-directed Translation from Standard ML into Java. Work in progress - manuscript available at `http://www.dcs.ed.ac.uk/home/cdw/papers/`, November 1998.

[Wal99] Chris Walton. Abstract Machines for Memory Management. Technical Report ECS-LFCS-99-410, LFCS, Division of Informatics, University of Edinburgh, June 1999.

[Wal00] Chris Walton. *An Abstract Machine for Memory Management*, chapter 10, pages 88–96. Trends in Functional Programming. Intellect, November 2000.

[WF94] Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, November 1994.

[Wil92] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of the International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St. Malo, France, September 1992. Springer-Verlag.

[WKG98] Chris Walton, Dilsun Kırlı, and Stephen Gilmore. An Abstract Machine for Module Replacement. In Stephan Diehl and Peter Sestoft, editors, *Proceedings of the Workshop on Principles of Abstract Machines*, pages 73–87, Pisa, Italy, September 1998. Also published as Technical Report A02/98 Universität des Saarlandes.

[WKG00] Chris Walton, Dilsun Kırlı, and Stephen Gilmore. An Abstract Machine Model of Dynamic Module Replacement. *Future Generation Computer Systems*, 16(7):793–808, May 2000.

[WM97] Chris Walton and Bruce McAdam. The C-LEMMA Memory Interface on the Cray T3D. Technical Report ECS-LFCS-97-362, LFCS, Division of Informatics, University of Edinburgh, July 1997.