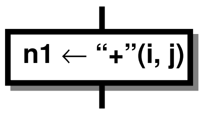
Chapter 9 Type Analysis

Type information plays a critical role in improving performance of object-oriented languages such as SELF. To obtain the maximum benefit from any type information the compiler can infer, the compiler uses sophisticated flow analysis to propagate type information through the control flow graph. This propagation is called *type analysis*, and is the subject of this chapter. To simplify the exposition, only type analysis for straight-line code without loops is discussed here; type analysis of loops will be described later in Chapter 11.

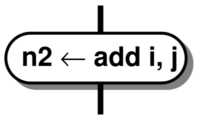
9.1 Internal Representation of Programs and Type Information

9.1.1 Control Flow Graph

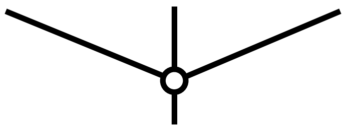
The compiler represents the method being compiled using a control flow graph data structure, with different kinds of nodes in the control flow graph for different kinds of operations. These nodes include high-level nodes such as message send nodes,



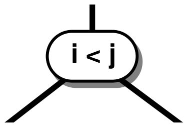
instruction-level nodes such as add instructions,



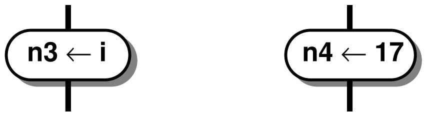
control flow nodes such as merge nodes



and conditional branch nodes,*



and bookkeeping nodes such as assignment nodes.

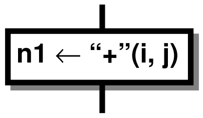


Most passes in the compiler involve some sort of traversal of this graph.

* In all diagrams in this dissertation, the "yes" or "success" arc of a branch node will exit on the left.

9.1.2 Names

The compiler uses *names* to capture data dependencies among the nodes. A name corresponds to either a source-level variable name (such as an argument or local variable) or a compiler-generated temporary name (such as a name referring to the result of a subexpression in the source code). The compiler treats both kinds of names in the same way. Names represent the flow of data through nodes in the graph by having some nodes bind names to computed results, with other nodes referring to bound names as arguments. For instance, the message send node

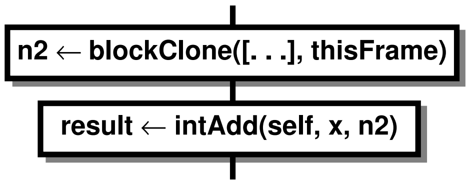


passes the data values bound to the names **`i`** and **`j`** as the receiver and argument to the **+** message, and binds the result data value to the (temporary) name **`n1`**.

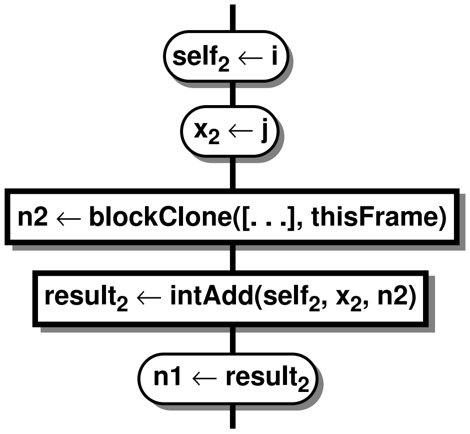
To illustrate control flow graphs, names, and inlining, consider the above message send node. If the compiler can infer that the type of the receiver **`i`** of the **+** message is, say, an integer, then it can lookup the **+** message for integers at compile-time and locate the following method:

```
+ x = ( _IntAdd: x IfFail: [ . . . ] ).
```

The compiler then can inline this method. Inlining a method entails constructing a new control flow graph for the inlined method

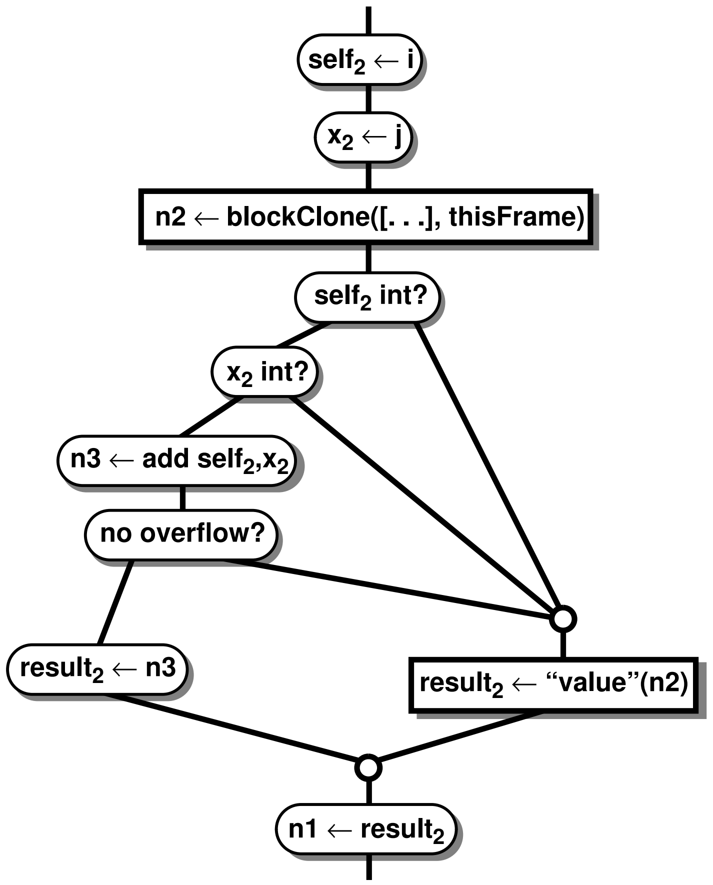

n2 ← blockClone([. . .], thisFrame)
result ← intAdd(self, x, n2)

and splicing this control flow graph into the main graph in place of the message send node. Name assignment nodes are inserted to assign the names of the actual receiver and arguments to the names used as the formals in the inlined control flow graph, and likewise to assign the name of the result used in the inlined control flow graph to the name of the result of the eliminated message send node.*


self₂ ← i
x₂ ← j
n2 ← blockClone([. . .], thisFrame)
result₂ ← intAdd(self₂, x₂, n2)
n1 ← result₂

* Names like **`self`**$_2$ and **`x`**$_2$ correspond to formals of inlined methods. New names are created for each inlined copy of a method.

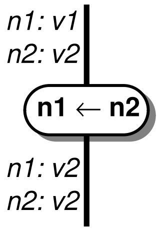
The compiler can also inline-expand the call to the **`intAdd`** primitive.



Most of the transformations of the control flow graph performed by the SELF compiler are of this flavor.

9.1.3 Values

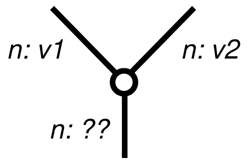
In our view, names are merely mechanisms for programmers and the compiler to refer to underlying data values and have no run-time existence. These underlying data values referred to by names are represented explicitly in the SELF compiler as *values*. Each value data structure in the compiler represents a particular run-time object. Many names may refer to the same value at a particular point in the program, and a single name may refer to different values at different points in the program. For example, after the assignment node



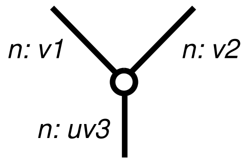
both **`n1`** and **`n2`** refer to the same value object (the value that **`n2`** referred to before the assignment node); **`n1`** may refer to a different value after the assignment than before the assignment. Since assignment nodes simply affect the compiler’s internal mappings from names to values, they do not directly generate any machine code.

The primary invariant relating names and values is that if two names map to the same value at some point in the program, then both names are guaranteed to refer to the same object at run-time at that point. On the other hand, if two names map to different values, then the compiler cannot tell whether the names will refer to the same object at run-time or not. Values are immutable; a new value is created whenever the compiler needs a representation for a run-time object that is potentially different from any other object. For example, the receiver and argument names of the method being compiled are initialized to new unique values, as are the results of non-inlined message sends, primitives, and integer arithmetic. References to the contents of assignable data slots in the heap (such as instance variable accesses) also are bound to new unique values.

Merges in the control flow graph pose an interesting problem for maintaining the invariant relating names and values. If a name is bound to a value **`v1`** along one predecessor branch of a merge node and bound to a different value **`v2`** along another predecessor branch, to what value should the name be bound after the merge?

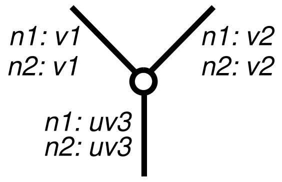


If either **`v1`** or **`v2`** is chosen, then for some paths through the program at run-time the compiler will have inferred incorrect information, potentially leading it to make incorrect optimizations that cause the optimized program to misbehave or even crash. Since neither incoming value is acceptable, a brand new value must be created to represent the run-time data value referred to by the name after the merge. The new values created by merge nodes are called *union values*.

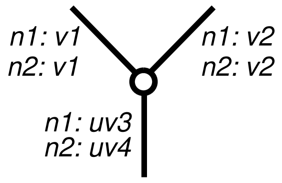


Union values are very similar to the ϕ-functions of SSA form, described in section 3.4.4.

Currently, each name that needs a new union value at a merge node is given a unique union value. A more accurate analysis would locate those names that for each predecessor branch are bound to the same value and assign them all the same new union value after the merge. For example, after the following merge node both **`n1`** and **`n2`** are guaranteed to refer to the same run-time object, and so should be given the same union value.

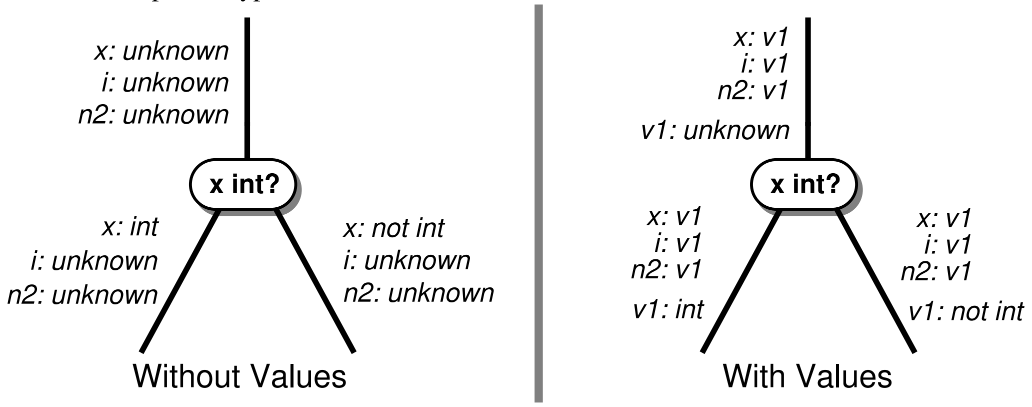


The SELF compiler's analysis currently is not sophisticated enough to recognize this situation, and so **`n1`** and **`n2`** each will be given its own new union value after the merge.



Values provide a better base than names for certain kinds of analysis and optimizations. As a consequence of aggressive inlining of user-defined control structures and operations, new names are created at a high rate, with many trivial assignment nodes introduced merely to assign one name to another (e.g., the name of an actual to the name of a formal). Values provide an explicit representation of the objects flowing through these names and as such are closer to the variables of a traditional language and compiler. Techniques and optimizations that are normally based on variable names in a traditional compiler, such as register allocation and common subexpression elimination, are more naturally based on values in the SELF compiler. (Common subexpression elimination with values will be described in section 9.6; global register allocation will be described in section 12.1.) In these respects, values serve purposes similar to those served by subscripted variables in SSA form (described in section 3.4.4). However, we feel that explicitly separating values from names, propagating values through the control flow graph independently from names and testing the values for equality, is simpler than first transforming the program into SSA form, with each name replaced with several subscripted names, then merging subscripted names into equivalence classes, and then testing the subscripted names for membership in the same equivalence class.

Values also improve the effectiveness of type analysis. Run-time type tests of a particular name, such as those implementing run-time type checking of the arguments of primitives or verifying guesses as part of type prediction (described in section 9.4), alter the type associated with the *value* that is bound to the tested name, instead of just the name itself as would happen in a simpler system that mapped names directly to types. This allows a single type test to alter the inferred type of several names, i.e., all those that are currently bound (aliased) to the same tested value. For example, without values, the type of only the tested value would be updated, while with values all aliased names will be updated with the improved type information.



Since many of these type tests occur deeply nested in inlined control structures and operations, the names that are used as part of the test are just local temporary names. Values therefore become critical for communicating this important type information beyond the local scope of the inlined control structure or operation. As shown in section 14.3, without values, SELF would run an average of 50% slower.

9.1.4 Types

Types are the primary data structures used by the compiler to represent type information and support various kinds of type-related optimizations such as compile-time message lookup and constant folding. A type describes a set of run-time objects usually sharing some common property. The particular kinds of sets the compiler is capable of describing concisely through types are motivated primarily by the optimizations currently performed by the compiler; new sorts of optimizations might require new kinds of type information to be represented and propagated through the control flow graph. The types currently used in the SELF compiler are described in the next few subsections.

9.1.4.1 Map Types

A *map type* specifies all objects that share a particular map, i.e., all objects in a single clone family.* This kind of type is perhaps the most important kind of type, since it is the most general type that still enables the compiler to perform message lookup at compile time and to perform type-checking of primitive arguments at compile time.

Map types are introduced by several sources:

- The type of **`self`** is a map type as a result of customization (described in Chapter 8).
- A run-time type test (such as testing whether an object is an integer) marks the tested object as being of a particular map type along the branch in which the test is successful.
- Some primitive operations are known to return objects of particular map types. For example, integer addition primitives are known to return integers if the primitive succeeds, and the **`_Clone`** primitive always returns an object with the same map as its receiver.

* For a class-based language this kind of type would specify all objects that are instances of a particular class and would be called a *class type*.

9.1.4.2 Constant Types

A *constant type* specifies a single object, i.e., a compile-time constant value. Constant types support the same sorts of optimizations as do map types, plus additional optimizations such as constant-folding of primitives.

Constant types are introduced by several sources:

- A literal in SELF source code is of constant type.
- The result of an inlined message that accesses a constant data slot (such as the **`true`** message) is of constant type.
- A run-time value test (such as testing for the **`true`** object at run-time) marks the tested object as being of a particular constant type along the success branch.

9.1.4.3 Integer Subrange Types

An *integer subrange type* specifies a range of integer values from a lower bound to an upper bound. Integer subrange types allow the compiler to perform some kinds of range analysis optimizations. For example, the compiler can eliminate an array bounds check when the range of the index is guaranteed to be within the bounds of the array. Similarly, the compiler can eliminate the overflow check from an integer arithmetic operation when the ranges of the two arguments prove that the result will not overflow the normal 30 bit integer representation. The compiler can even "constant-fold" an integer comparison when the ranges of the two arguments do not overlap. The integer map type and the integer constant types may be viewed as extreme cases of integer subrange types, although they are represented more concisely than other integer subrange types.

Integer subrange types are introduced by several sources:

- The result of a successful integer arithmetic primitive is an integer subrange (or an integer constant or the integer map type) computed from the ranges of the arguments to the operation.
- An integer comparison operation narrows the types of its arguments depending on the outcome of the comparison. For example, when comparing **`i < j`**, along the true branch the compiler can lower the upper bound of the type of **`i`** to be one less than the upper bound of the type of **`j`** (and similarly raise the lower bound of the type of **`j`**); along the false branch the analogous narrowing can occur. This narrowing can convert an integer map type into an integer subrange type.
- Some primitive operations are known to return integers within a particular range. For example, the array size primitive returns only non-negative integers less than some upper limit bounded by the maximum size of the heap.

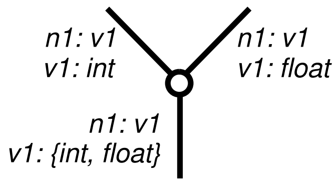
9.1.4.4 The Unknown Type

The *unknown type* specifies all possible objects and so conveys no information to the compiler. The compiler associates the unknown type with incoming arguments to the method being compiled (no customization is performed for arguments), the results of non-inlined messages, the contents of assignable data slots in the heap (e.g., instance variables), and the results of some primitive operations.

9.1.4.5 Union Types

A *union type* specifies the union of the objects specified by its component types. If a name is known to be of a particular union type, the contents of the variable at run-time could be any of the objects possible for any of the component types. If all the component types are covered by the same map type (i.e., all the component types specify objects in a single clone family), then the union type allows the same sorts of optimizations as a map type. It is more common, however, for the component types to be of different clone families, and so a union type typically provides less information than a map type but more information than the generic unknown type. Union types guide both run-time type casing (described in section 9.3) and splitting (described in Chapter 10).

Union types are created primarily as a result of merges in the control flow graph in which one value is associated with different types on different predecessor branches, analogously to union values. For example, if some value is associated with the integer map type along one predecessor branch of a merge, and associated with the floating point map type along the other predecessor branch, after the merge the value will be associated with the union type whose components are the integer map type and the floating point map type.



Union types are also created as the result of certain primitive operations. For example, the type of the result of a comparison primitive (along the success branch) is the union of the **`true`** constant type and the **`false`** constant type: *{true, false}*.

9.1.4.6 Exclude Types

An *exclude type* specifies a set of component types that the associated value is known *not* to be. Exclude types are introduced as a result of unsuccessful run-time type tests, such as recording that an object cannot be an integer along the failure branch of an integer type test. The compiler uses exclude types to avoid repeated tests for possibilities that are guaranteed not to occur.

Exclude types are not as expressive as would be full-fledged *difference types*. A difference type would specify those objects that were in one type but not in another, i.e., the set difference of the objects specified by two types. An exclude type is equivalent to the difference of the unknown type and the union of the excluded types:

<not t1, t2, t3> ⇔ *<unknown type> - <{t1, t2, t3}>*

Only differences from the unknown type can be expressed with an exclude type; an exclude type cannot express differences from some more precise type. For example, the current type system cannot record that some possibilities out of a map type are excluded, such as after a failed run-time value test of an object whose map type was known. A full-fledged difference type could describe this type as the difference between the known map type and the union of the excluded possibilities within the map type.

Unfortunately, generalizing exclude types to difference types would create new problems. With general difference types, a single type could be represented in multiple ways, significantly complicating type equality testing and other sorts of comparisons on types, such as whether one type covers another. This problem already exists in the current type system to some extent for integers, since an integer subrange type can represent the same type as a union of adjacent or overlapping integer constant types and/or integer subrange types. For example, the following types should all be considered equivalent:

{0, 1, 2, 3, 4, 5}	a union of integer constant types
{[0..2], 3, [4..5]}	a union of integer subrange and integer constant types
[0..5]	an integer subrange type

Adding general difference types exacerbates this problem, since now the following difference type also is equivalent to the above types:

```
int - {[minInt..-1], [6..maxInt]}
```

Even now the current SELF compiler does not always detect that types such as the first three types above are equivalent.[*] We chose not to worsen this situation, and consequently do not include fully general difference types in our type algebra. Luckily, exclude types seem to be sufficient in practice, since only rarely does the compiler perform value tests on objects whose map was already known. Nevertheless, the SELF compiler eventually should include some generalized approach to equality and other type comparisons in the presence of union types, difference types, and integer subrange types, perhaps by defining some canonical representation of sets of integers and translating into the canonical form prior to comparing types.

[*] This inaccuracy does not lead to incorrect compilation, since erring on the side of treating two things as of different types is safe, but it can lead to poorer generated code.

9.2 Type Analysis

To perform optimizations like compile-time message lookup and elimination of run-time type checks, the compiler needs to be able to determine the type associated with a name at a particular point in the program. To support this determination, the compiler maintains a mapping from names to values and a mapping from values to types. These mappings are propagated through the control flow graph as type analysis proceeds.

When performing type analysis, the compiler visits each node in the control flow graph in topological order. Each control flow graph node implements its own type analysis routine.* This routine typically examines the type mappings propagated from the node's predecessor(s), performs any optimizations of the node based on the type information, and finally produces new type mappings for the node's successor(s). The following subsections describe some node-specific type analysis operations in more detail. Discussion of type analysis in the presence of loops will be deferred until Chapter 11.

9.2.1 Assignment Nodes

An assignment node alters the name/value mapping of the assigned name. In the name/value mapping after the assignment node, the value associated with the assigned name (the "left-hand side") is the same as the value associated with the assigned-from name (the "right-hand side"). All other bindings are unaffected.

9.2.2 Merge Nodes

A merge node combines the name/value and value/type mappings from its predecessors to form a new pair of mappings after the merge. New union values and union types may be constructed.

9.2.3 Branch Nodes

A branch node makes two copies of the mappings, one for each successor branch, so that subsequent alterations to a mapping along one successor branch do not affect the mapping along the other successor branch.

9.2.4 Message Send Nodes

A message send node looks up the type bound to the receiver name. If this type is a map type (or more specific than a map type), then the compiler performs compile-time message lookup. If the lookup is successful and the compiler elects to inline the target method, then the message send node is replaced with a control flow graph representing the inlined target method; type analysis then begins to analyze the new inlined method. If the message is not inlined, then a new value is created to represent the result of the message send. The name of the message result is bound to this new value and the new value is bound to the unknown type. These altered mappings are then passed on to the message send's successor.

9.2.5 Primitive Operation Nodes

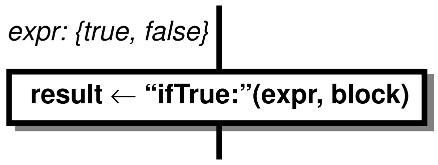
A primitive operation node first checks to see whether it can be inlined (whether the compiler has the implementation of the primitive built-in). If so, then the primitive operation node is replaced with nodes that implement the primitive in-line. The type information associated with the arguments to the primitive can be used to optimize inlined primitives, such as by eliminating unnecessary argument type checks, overflow checks, or array bounds checks. If the primitive is inlined, then type analysis continues with the first node of the primitive. Otherwise, a new value is created for the primitive's result and the primitive's result name is bound to this new value. The result value is in turn bound to either the unknown type or to some more precise type depending on the primitive and its argument types. The altered mappings are then passed on to the primitive's success to continue type analysis.

* The compiler is implemented in C++. Each control flow graph node is an instance of a class, with different classes for different kinds of control flow graph nodes. All control flow graph nodes define a virtual function which implements the node-specific type analysis.

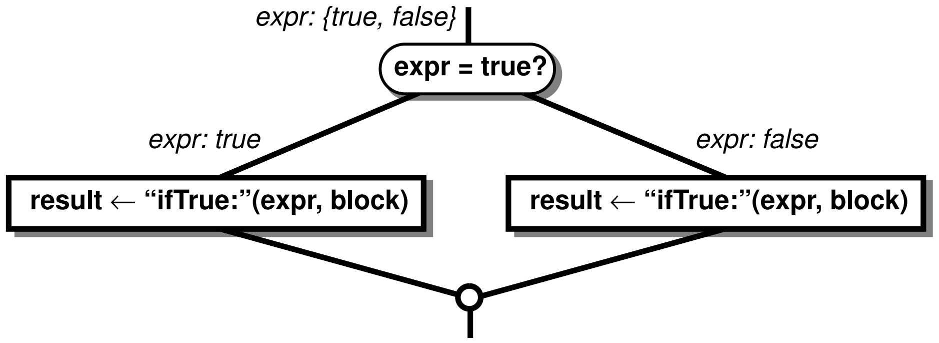
9.3 Type Casing

When the compiler can infer that the type of the receiver of a message is covered by a map type, the compiler can perform message lookup at compile-time, statically-binding and inlining the message. However, the type of a receiver frequently will not be a simple map type but instead will be a union of several different types. This can occur as a result of a primitive known to the compiler to return one of several different types (such as comparison primitives returning either **`true`** or **`false`**) or after a merge in the control flow graph where a single name was associated with several different types before the merge.

Type casing is a simple technique for optimizing message sends where the receiver is bound to a union type. For example, the **`ifTrue:`** message frequently is sent to an expression whose type is the union of the **`true`** constant type and the **`false`** constant type:



Optimizing such messages with type casing involves testing for each element type in the union and then branching to code specific to that type. Specifically, for each component of the union type that is covered by a map type, the compiler inserts a run-time type test before the message that checks for that type and in the successful case branches to a copy of the message. In the **`ifTrue:`** example, the compiler would perform type casing by inserting a run-time check for **`true`** that branched to a copy of the **`ifTrue:`** message:


expr: {true, false}
expr = true?
expr: true
expr: false
result ← "ifTrue:"(expr, block)
result ← "ifTrue:"(expr, block)

Each copy of the type-cased message now can be statically-bound and inlined away, since the type of the receiver is known after a successful run-time type test. Furthermore, if all the components of the union type are covered by map types (such as in the **`ifTrue:`** example), then the last component of the union type does not need a run-time type check, since the other failed type checks will have excluded all other possible types. If on the other hand some components of the union type are not covered by a single map type (such as if the union type contains the unknown type as one of its components), or if the message cannot be inlined for some map types in the union, then a final dynamically-dispatched version of the message will be needed to handle the remaining case(s). Control flow re-merges after the copies of the message.

Type casing is a simple technique to take advantage of union type information. It has the effect of transforming a single polymorphic message into several monomorphic messages that can be further optimized. As such, it is an example of the general theme in the SELF compiler of trading away space for improved run-time performance. Type casing is also very similar to case analysis as performed in the TS compiler for Typed Smalltalk (see section 3.1.3). However, type casing incurs run-time overhead with the extra type tests for various component types of the union. While the resulting code is typically faster than the original message send (sometimes much faster, such as in the **`ifTrue:`** example above), it is not as fast as is frequently possible. Chapter 10 discusses *splitting*, a more sophisticated technique for exploiting union type information created by merges in the control flow graph without run-time overhead. Where the compiler cannot apply splitting, such as for unions created as the result of primitive operations, it falls back to this type casing technique.

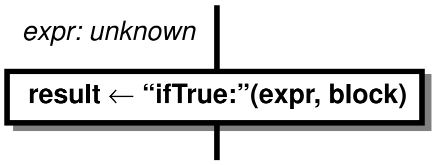
Type casing illustrates that a union type provides more information than just specifying the set union of the objects specified by its component types. The divisions of the various component types is also important, since it is these component types that drive type casing. For example, if prior to a merge node a name is bound to the integer map type along one predecessor and to the unknown type along the other predecessor, as part of type analysis the compiler will bind the name to the union of these two types after the merge: *{integer, unknown}*. A naive implementation would "simplify" this union type by noting that the unknown type covers the integer map type, and so the integer map type could be removed from the union without altering the set of objects specified by the type: *{unknown}* or simply *unknown*. However, such a simplification would lose a significant amount of important information. The compiler no longer would have the information that integers are a likely component of the type, and so the compiler no longer could separate out the integer case via type casing. If faced with a blank unknown type, the compiler would have no information upon which to decide that some subset of the possible types would be worth type casing for. Accordingly, union types in the SELF compiler are never "simplified" by eliminating component types covered by another component type, to preserve as much useful information as possible.

9.4 Type Prediction

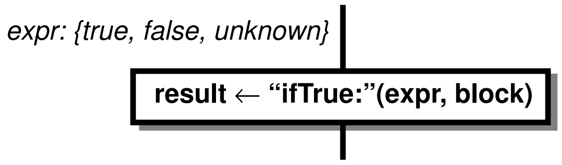
Customization and type analysis enable the compiler to infer the representation-level types of many expressions, which in turn enables the compiler to statically-bind and inline many messages and optimize away many run-time type checks. However, many messages remain whose receiver types cannot be inferred using only customization and type analysis. For example, the types of arguments are unknown (the compiler currently does not customize on the types of arguments), as are the types of the contents of assignable data slots in the heap (e.g., instance variables), so messages sent to these objects cannot be inlined using only customization, type analysis, and type casing.

To enable the compiler to infer even more type information, the SELF compiler performs *type prediction*. If the compiler cannot infer the type of the receiver of a message using customization or type analysis, then the compiler tries to use the *name* of the message as a hint about the likely type of the receiver. For example, the compiler predicts that the receiver of a **+**, **<**, or **to:Do:** message is likely to be an integer, the receiver of an **at:** message has a good chance of being an array, and the receiver of an **ifTrue:** message is almost certainly either **true** or **false**. These predictions are embedded in the compiler in the form of a small fixed table mapping message names to likely receiver types. A message whose name is not in the table is not type-predicted, and so must be implemented as a full message send in the absence of other optimizations.

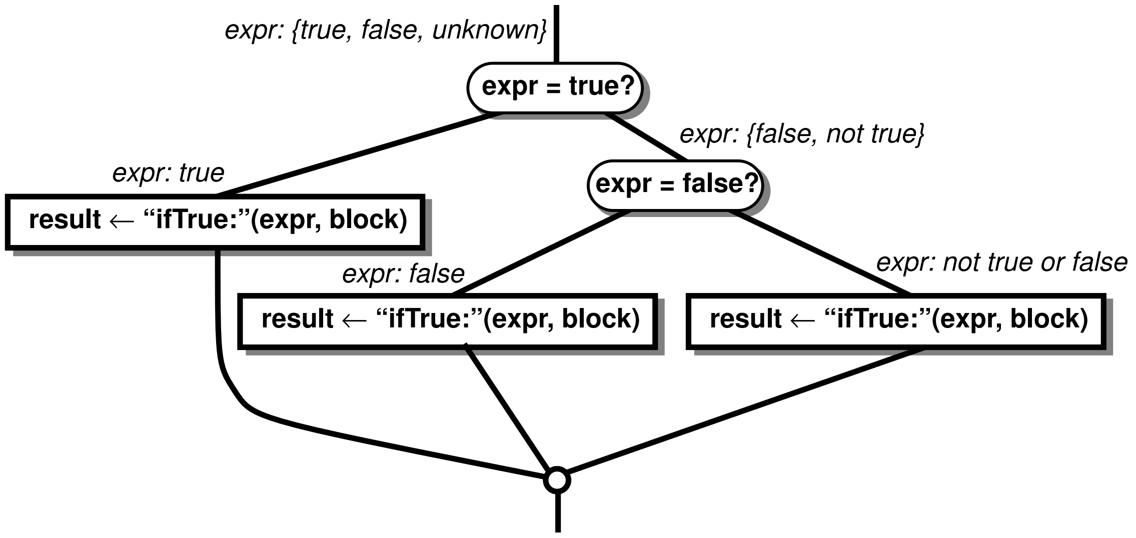
The compiler uses the predicted type(s) to transform the original receiver type into a union type, one of whose components is the original receiver type and whose other component(s) are the predicted type(s). For example, if the **ifTrue:** message is sent to an expression whose type is unknown:



then the compiler can perform type prediction and replace the receiver type with one that contains the **true** and **false** constant types:



Since the original type is still part of the union type, the receiver's type remains just as general as it was before type prediction. However, the form of the receiver's type is now suitable for further optimization via type casing: a run-time type test is inserted to check for each predicted type and then branch to a separate statically-bound copy of the message suitable for inlining; a fall-back dynamically-bound version of the message will remain in case none of the predictions are correct. In the **`ifTrue:`** example, after type prediction the compiler would apply type casing and insert run-time checks for **`true`** and **`false`** that each branched to a separate copy of the **`ifTrue:`** message:



expr: {true, false, unknown}
expr = true?
expr: {false, not true}
expr: true
expr = false?
result ← "ifTrue:"(expr, block)
expr: false
expr: not true or false
result ← "ifTrue:"(expr, block)
result ← "ifTrue:"(expr, block)

Two copies of the **`ifTrue:`** message can be inlined away, since their receiver types are both compile-time constants; the third copy remains dynamically-bound since its receiver type is unknown. Control flow re-merges after the type prediction and type casing transformations.

If the rate of successful prediction is high, the performance of a predicted message can be much faster than the original unpredicted message. The cost of a type test is less than the cost of a dynamically-dispatched procedure call, and inlining predicted messages can lead to opportunities for additional time-saving optimizations. Of course, since the unsuccessful branch is slowed down by an extra run-time type test, using type prediction in cases with a low success rate can actually slow down the overall performance of the system.

In the current SELF system, type prediction has a high success rate. In the benchmarks used to measure the performance of SELF, almost all predictions are correct. This is because the benchmarks were originally written in traditional languages such as Pascal and BCPL, and the data types predicted using type prediction (integers, booleans, and arrays) are those that are normally the only ones used in traditional languages; the only mispredictions occur in benchmarks translated from Lisp which overload **=** to compare both integers and **`cons`** cells. However, even in large Smalltalk systems, the receiver of a message like **+** is an integer 90% to 95% of the time [Ung87], so type prediction is useful even for programs written in a heavily object-oriented style. As reported in section 14.3, type prediction speeds SELF programs by a factor of 3 on average, with object-oriented SELF programs benefitting almost as much as more traditional, numeric benchmarks.

Type prediction is similar to (and was inspired by) the technique in Smalltalk-80 systems that hard-wires the implementations of certain common messages into the compiler, as described in section 3.1.1. Both insert run-time type tests to verify static predictions embedded in the compiler. However, unlike Smalltalk's hard-wiring, type prediction does not embed the *definition* of predicted messages into the compiler (since inlining is used instead), nor does the compiler impose any *restrictions* on the use of predicted messages such as **`ifTrue:`** and **==**. Programmers are always free to change the definitions of predicted messages and to add new definitions that did not exist when the compiler was implemented. Type prediction coupled with inlining enables the SELF implementation to achieve the same run-time performance as hard-wired messages and still preserve SELF's pure message passing model.

Type prediction as currently implemented is a static technique: the message names and predicted receiver types is fixed in the compiler and cannot be changed by users. A better technique would be *dynamic type prediction*, where the message names and predicted receiver types would automatically adapt to the SELF source code currently in use. For example, dynamic profile data could be used to augment or replace the static table built into the compiler. We are actively investigating techniques that would make type prediction more adapting to changing usage patterns [HCU91].

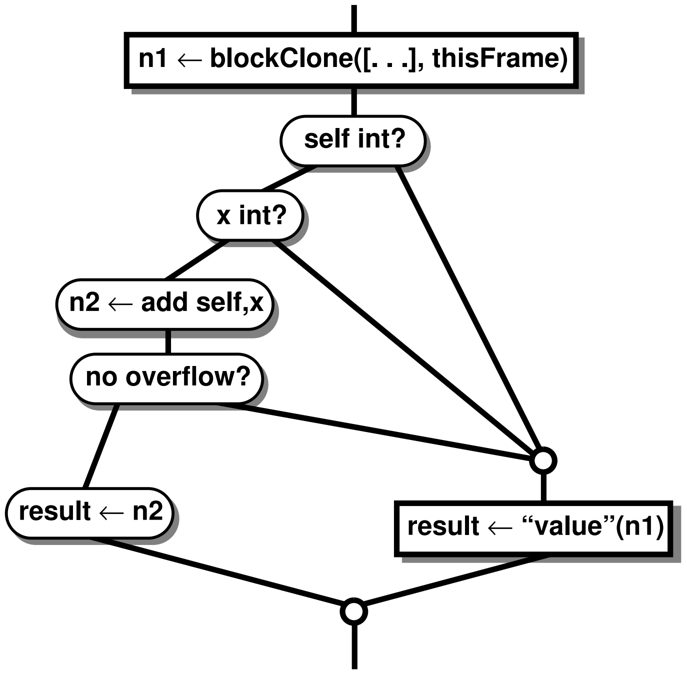
9.5 Block Analysis

Blocks are very common in SELF code, primarily because of their central role in the implementation of user-defined control structures. In a straightforward SELF implementation, each invocation of a control structure like **`to:Do:`** (SELF's version of a traditional **`for`** loop) would involve creating several blocks at run-time and invoking these blocks repeatedly during the execution of the loop. This approach to compiling user-defined control structures and blocks would never be competitive in run-time performance with traditional languages based on built-in control structures, where the compiler can generate only a few instructions to implement a control structure. Consequently, much of the SELF compiler's efforts are directed towards eliminating the overhead of user-defined control structures and blocks.

9.5.1 Deferred Computations

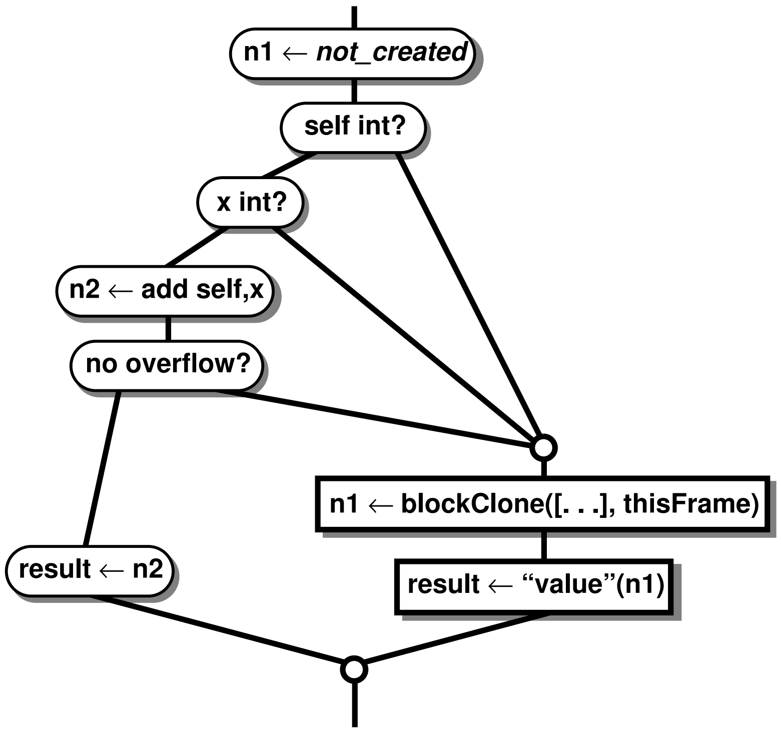
Several important block-related optimizations have already been mentioned in section 7.1: inlining statically-bound block **`value`** methods (and eliminating block creation operations if all uses of the block are eliminated) and preferentially inlining methods with block arguments to increase the likelihood of the block arguments getting inlined away. However, many blocks may have a few remaining uses that cannot be eliminated, thus requiring that the block be created, even if those uses are only on control flow paths that are executed rarely.

For example, consider an inlined integer addition primitive operation that is passed a failure block. If the primitive fails, then the block is sent the **`value`** message, and so if this message is not inlined (as it is not in the current SELF system) one use of the block remains and the block creation code cannot be completely eliminated.



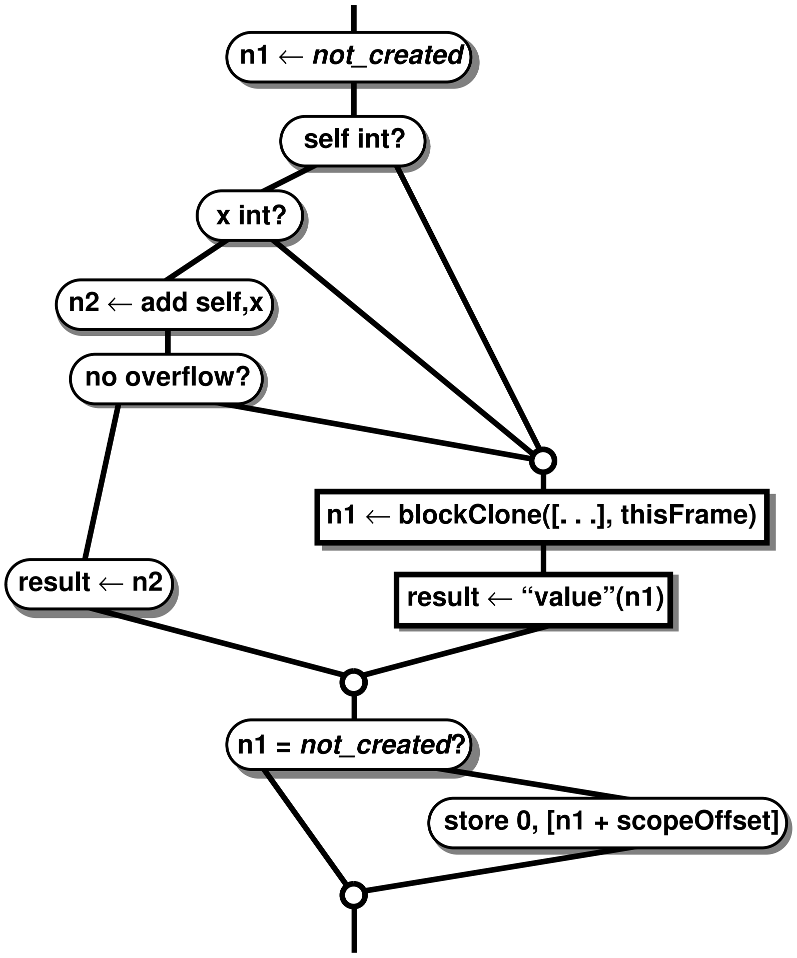
However, since most primitive invocations do not fail, the large majority of the invocations of the primitive do not use the created block. The overhead of creating a block for every primitive invocation, especially ones as simple as integer arithmetic, can quickly bring a system to its knees.

To limit the overhead of the block creation to just those parts of the program that need the block, the compiler *defers* the creation of blocks until they are actually needed as a run-time value. Then only those paths through the control flow graph that need the block will pay the expense of creating the block; other paths are not penalized.

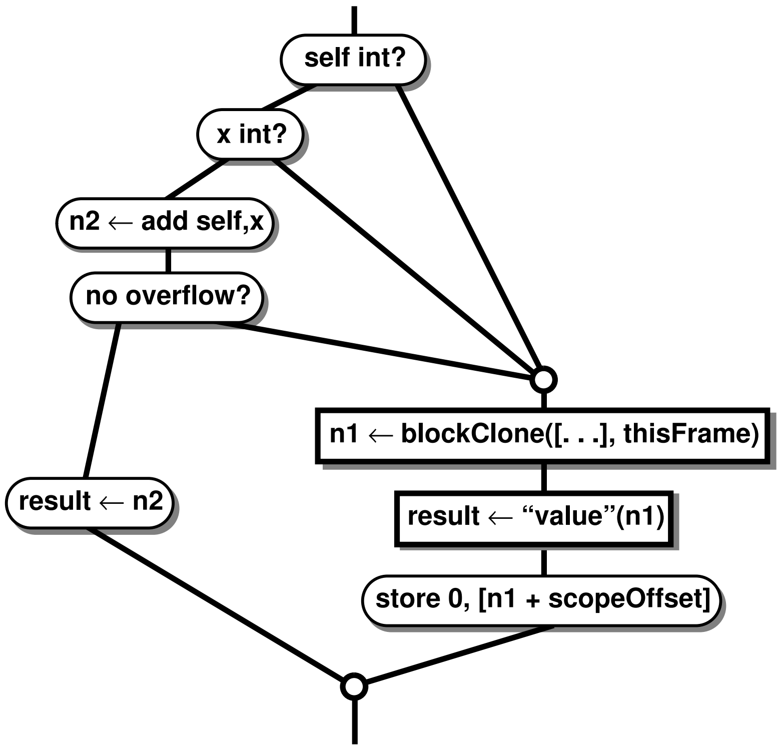


The compiler needs some way to tell whether a block has been created or not at some point in the program. This information is encoded in the type associated with the block value: a block's type is either a *deferred block type* or a *created block type*. In place of the original block creation code, the compiler generates an assignment node that binds a fresh value object to the deferred form of the block type. Any later uses that require the block to be a real run-time value, such as a non-inlined message send with the block as an argument, check whether the type of the block is deferred and if so insert additional nodes in the control flow graph before the use to create the block object at run-time. The block creation node rebinds the value object representing the block from the deferred block type to the corresponding created block type; the name/value binding of the created block remains unchanged so that all names aliased to the same block value simultaneously see that the block is now created, thus avoiding duplicate block creations.

Deferred computations such as block creation are handled at the same time other type analysis and optimizations are performed so that whether or not an expensive computation has been performed can influence the control flow graph that gets built. This is especially important for optimizing block zapping code (see section 6.3.2) out of those paths of the control flow graph where the compiler knows the block is never created. In an earlier version of the SELF compiler, deferred computations were handled in a later pass of the compiler after the bulk of the control flow graph transformations had been performed. This design led to poor performance, since some extra run-time tests had to be inserted to check whether the block creation had been performed before zapping the block. For example, in the following graph which includes the code to zap the failure block, even though the block creation is deferred to the primitive failure branch, the block zapping code remains even for paths where the primitive succeeds.



Handling deferred computations as part of type analysis allows other techniques that cooperate with type analysis, such as splitting (described in Chapter 10), to help optimize the treatment of blocks and block zapping so that only blocks that have been created need zapping code. This enables the block zapping code to be executed only along the branches where the block is created.



Optimizing away block creations when unnecessary, and deferring remaining block creations until just prior to their uses, is one of the most important optimizations performed by the SELF compiler. As reported in section 14.3, without deferred block creation SELF would run an average of 4 times slower, and more that 10 times slower for programs rife with user-defined control structures such as **`to:Do:`** loops.

Other computations such as arithmetic are also deferred until their first uses. All that is required of a deferred computation is that it have no externally-visible side-effects. Opportunities for deferring computations other than block creations occur far less often, however, and do not produce the same sorts of dramatic speed-ups as when deferring block creations.

9.5.2 Analysis of Exposed Blocks

Blocks support lexical scoping of local variables: a nested block can contain accesses and assignments to arguments and local variables in a lexically-enclosing scope. These accesses and assignments are called *up-level accesses* and *up-level assignments*. If a block’s **`value`** method is inlined, up-level accesses and assignments in the **`value`** method can be treated as local accesses and assignments, and type analysis can be used to infer the types of these variables. An inlined block **`value`** method’s non-local return can also be implemented as a direct branch to another part of the control flow graph if the block’s outermost lexically-enclosing scope is also inlined within the same compiled method.

However, when a block cannot be optimized away and is instead passed as the receiver or argument to a non-inlined message, the compiler must become more conservative. Since the non-inlined message might invoke the block, the types inferred for any local variables potentially up-level assigned from within the block’s **`value`** method must be weakened to include the unknown type; the current SELF compiler does no interprocedural analysis to infer the types of expressions up-level assigned from within non-inlined blocks. Also, since the non-inlined method might store the block in a long-lived global data structure, all subsequent non-inlined messages must be assumed to invoke the block, again requiring the types of up-level assigned variables to be weakened. We call blocks that might be invoked by other methods at any call site *exposed blocks*, since they have been exposed to the outside world and are no longer under tight control.

Since exposed blocks dilute the type information of potentially up-level assigned local variables, the compiler works hard to limit the number of blocks that must be treated as exposed. In addition to maintaining the name/value and value/type bindings during type analysis, the compiler maintains the set of blocks that have been exposed. A block is added to the current exposed blocks set if and when it is passed out at a non-inlined message send or stored into a data structure in the heap. In addition, all blocks up-level accessible from a newly exposed block must also be added to the exposed blocks set, since they might be accessed and invoked whenever the original block is invoked. A block is removed from the exposed blocks set once its lexically-enclosing scope returns (since the block will be zapped and unusable). At merge points, the compiler unions together the exposed blocks sets of the merge's predecessors to form the set of exposed blocks for the merge's successor, much as the compiler forms union types at merge nodes.

An exposed block set is used at a non-inlined message send node to alter the type bindings of all potentially up-level assigned variables of all exposed blocks in the set. When calculating the mappings for the message send's successor, each potentially up-level assigned name is rebound to its own new, unique value object, modelling an assignment to the name of an unknown object from within the exposed block.

The compiler must also somehow generalize the type associated with the new value object, since if the assignment does occur the compiler does not know what type of object will be assigned to the local variable. A simple strategy would simply bind the new value object to the unknown type. Unfortunately, such a naive approach would sacrifice any type information the compiler had inferred about the local variable prior to the message. The SELF compiler therefore tries to limit the damage to type information caused by a potential up-level assignment based on two heuristics. First, many potential up-level assignments will not actually be performed, so any information accumulated about the contents of the local variable would still be true after the message send. Second, of those assignments that are performed, the type of the variable after the assignment is likely to be similar to the type of the variable before the assignment, say in the same clone family; programs do not normally assign completely unrelated objects to the same local variable.

To exploit these trends, the SELF compiler assigns the type of a local variable after a potential up-level assignment as the union of the unknown type (in case the assignment actually occurs) and the original type of the local variable before the message send, generalized to the enclosing map type (in case the assignment does not occur or an assignment to a member of the same clone family occurs). For example, if the type of a potentially up-level assigned variable were *{[0..5], float}* before the message send, after the message send the type would be changed to *{unknown, integer, float}*. This union type information then can be exploited using type casing (see section 9.3). If the local variable is not assigned, or is assigned a member of the same clone family, this strategy incurs the overhead of only a type test upon subsequent accesses to the local variable, rather than requiring a full message send as would the naive strategy.

Exposed block sets improve the quality of type analysis by limiting the damage of up-level assignments to those parts of the control flow graph where blocks were actually created and exposed to the outside world. Since many blocks are only created and exposed along relatively uncommon branches, such as primitive operation failure blocks, the effect of these exposures is limited to those parts of the graph. This containment allows the common-case branches to execute without the conservative assumption that some local variables might have been assigned. In practice, exposed block analysis is very effective; as shown in section 14.3, with exposed block analysis SELF programs run almost 50% faster than they would if the compiler treated all blocks as exposed.

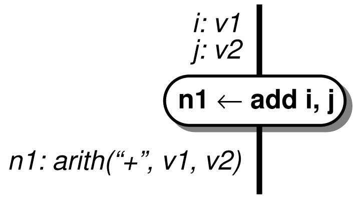
9.6 Common Subexpression Elimination

The compiler performs *common subexpression elimination* (*CSE*) as part of type analysis. CSE eliminates redundant computations by detecting when a computation has already been performed and its earlier result can be reused. The SELF compiler performs CSE on two kinds of computations: arithmetic calculations and memory references (loads and stores).

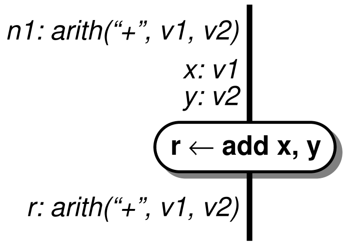
9.6.1 Eliminating Redundant Arithmetic Calculations

The compiler discovers redundant arithmetic calculations by searching the name/value mappings for an existing value that is the same as the result value of the potentially redundant calculation. If such a value already exists in the name/value mapping, then the compiler replaces the arithmetic calculation with a simple assignment of a name bound to the existing value to the name of the result of the eliminated calculation. To support equality comparisons of values, values representing the results of arithmetic calculations are structured, containing subcomponents for the receiver and argument values and the kind of arithmetic calculation. Two arithmetic values are guaranteed equal if and only if they have equal subcomponents.

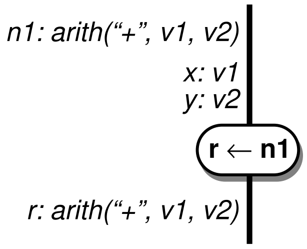
For example, the following **add** control flow graph node produces a structured result value:



If some later node in the graph calculates the same value:



then the compiler can replace the second redundant calculation with a simple assignment node:



The compiler typically can avoid generating code for these assignments by arranging that all names related by simple assignment are allocated to the same register (register allocation is described in section 12.1).

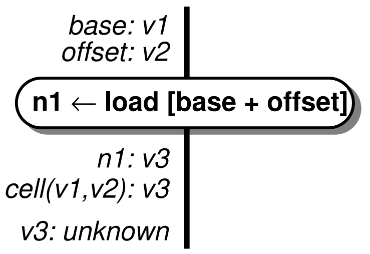
Currently the compiler detects equal calculations only if the operations are equal, the operands are equal, and the order of operands is the same, i.e., if the operation/operand trees are isomorphic. More sophisticated value equality systems would take into account arithmetic identities, such as the commutative property of addition and the relationships between addition and subtraction. This weakness in the current SELF compiler's rules actually makes a difference in the quality of generated code, although not a large difference. Additionally, values could be extended to enable equality testing even in the presence of conditional branches; the algorithms associated with subscripted names and SSA form, described in section 3.4.4, support such flow-sensitive equality testing. It remains an open question, however, how much practical benefit would be received from such additional analysis.

As reported in section 14.3, CSE of arithmetic expressions has a relatively minor effect on run-time performance, usually less than a 5% improvement in performance. Redundant arithmetic computations probably are less common in SELF programs than in traditional programs, in part because array accesses in SELF do not require multiplication of the array index by a scaling factor as in other languages, since SELF's tagged integer representation is already appropriately scaled for indexing into SELF's built-in one-dimensional object vectors. Also, other calculations that could be eliminated as redundant currently are not because of limitations in the garbage collector's treatment of derived pointers.

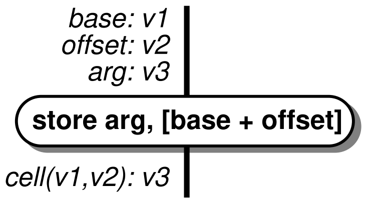
9.6.2 Eliminating Redundant Memory References

Redundant memory fetches and stores are detected in a fashion similar to detecting redundant arithmetic calculations. The compiler maintains a mapping from *cells* to values that is propagated through the control flow graph as part of type analysis. Cells are the compiler's internal representations of memory locations in the heap, such as assignable data slots (instance variables), array elements, and the lexically-enclosing frame pointers of blocks. The value object associated with the cell represents the current contents of the cell. A cell is "addressed" by two component values: a base value and an offset value. The base value is the value of the object being accessed by the memory reference; the offset value is either a constant (for fixed-offset memory references such as accesses to assignable data slots and frame pointers) or a normal computed value (for computed indexes into arrays). Two cells are guaranteed to refer to the same physical memory location if their corresponding base values and offset values are equal. Of course, two cells with different base and offset values *might* still refer to the same memory location, however, since two values that are not guaranteed to refer to the same run-time object might still do so (see section 9.1.3).

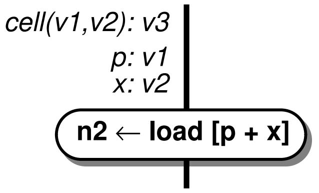
CSE of memory references works in much the same way as CSE of arithmetic calculations. Memory load instructions construct a new value object representing the contents of the cell and a new cell object addressed by the base and offset of the load instruction to the new contents value object. Three bindings are added to the mappings maintained by type analysis: a mapping from the name of the result of the memory load to the new value object is added to the name/value table, a mapping from the value to the unknown type is added to the value/type table, and a mapping from the new cell to the new value is added to the cell/value table.



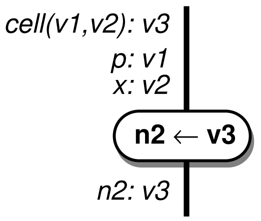
Memory store instructions similarly add a binding from a new cell object addressed by the base and offset of the store instruction to the value bound to the name of the object stored into the addressed memory location.



If at a later memory fetch node the same cell is already in the cell/value mapping table



then the compiler can replace the redundant memory fetch node with a simple assignment node, binding the result name of the redundant memory reference to the value associated with the existing cell.



In addition to eliminating an unnecessary memory reference, this optimization also preserves any type information the compiler has been able to infer about the contents of the memory cell. In some cases, the benefits from preserving this type information outweigh the benefits from merely eliminating an instruction. If CSE were performed at a later stage in the compilation process, as is done in other compilers for traditional languages, then this ability to preserve type information of memory cells would be lost. In effect, CSE of memory accesses supports type analysis of locations in the heap.

The compiler can eliminate a memory store node if the cell being assigned is already in the cell/value mapping and if the current contents value bound to the cell in the mapping is the same value as the one being stored. Otherwise, the store cannot be eliminated, and the compiler updates the cell/value mapping after the store node to reflect the compiler's knowledge about the cell's new contents. Additionally, all cells already in the mapping that *might* refer to the same memory location as the stored cell (i.e., any potentially aliasing cells in which the base and offset values *might* be the same) must be removed from the cell/value mapping, since their contents are now ambiguous. Similarly, non-inlined message sends must be assumed conservatively to assign to all global heap cells that could be assigned. Therefore, at each non-inlined message send, all cell/value bindings for cells that could be changed by a SELF program (including instance variables and array elements but excluding frame pointers and array sizes) must be removed from the cell/value mapping; such cells could be called "exposed" cells in analogy with exposed blocks.

To avoid losing precious type information, such potentially modified cells are treated in much the same way as potentially up-level assigned local variables (described in section 9.5.2). Each potentially-modified cell is given a new unique value object and added back into the cell/value mapping; the new value object is bound to the union type constructed by generalizing its previous inferred type to the enclosing map type and combining this type with the unknown type. In this way, the damage to type information from assignments to potential alias cells and potential assignments to "exposed" cells can be limited.

CSE of memory cells is used for another purpose in the SELF compiler: eliminating unnecessary array bounds checks. At each array access the compiler checks to see whether the cell corresponding to the accessed array element is already in the cell/value binding. If so, then the compiler omits the code that would have checked whether the array access was in bounds. This optimization is legal because the program must have been able to access the array cell before without error, since the cell is available; the bounds check must already have been performed as part of the previous access. Theoretically, this optimization should be unnecessary, because the compiler should be able to eliminate the bounds check more directly by remembering that the bounds check already had been performed for the same array index. However, the current implementation of the SELF compiler does not record this information. Checking for CSE of memory cells is therefore a cheap way of eliminating some redundant array bounds checks without much additional mechanism. In the future, however, the SELF compiler should include enough information to be able to determine when array bounds checks have already been performed (or, even better, when no array bounds checks are required at all at run-time), and this technique will be removed as redundant.

As reported in section 14.3, common subexpression elimination of redundant memory references is more effective for SELF than common subexpression elimination of redundant arithmetic. While the average performance improvement from CSE of memory references is around 5%, some benchmarks speed up by more than 30% from this technique. The effect of being able to track type information through assignments and subsequent fetches from memory cells accounts for a sizable fraction of the total contribution of CSE of memory references; eliminating array bounds checking using available cell information accounts for a smaller fraction.

9.6.3 Future Work: Eliminating Unnecessary Object Creations and Initializations

Ideally, the compiler could eliminate some object creations and stores if all uses of the object (such as memory fetches out of the created object) were eliminated. As an example of a situation where such throw-away objects get created, a quadratic formula function might return multiple roots by creating an object with a pair of assignable data slots, store the result roots into the object, and then return the object:

```
quadraticFormulaA: a B: b C: c = ( | result. temp. |
  temp: (b squared - (4 * a * c)) sqrt.
  result: ( | r1. r2. | ) _Clone. "create an object to hold the roots"
  result r1: (b negate + temp) / (2 * a).
  result r2: (b negate - temp) / (2 * a).
  result ).
```

The caller of the quadratic formula routine would extract the roots out of the result object's data slots and then throw the result object away:

```
printRootsA: a B: b C: c = ( | result |
  result: quadraticFormulaA: a B: b C: c.
  (result r1 printString, ' & ', result r2 printString) printLine.
).
```

If the called routine were inlined into the calling routine, then the memory fetches that extract the data slots of the returned object would be optimized away as redundant, since the compiler would have recorded the earlier stores to the same memory cells. Thus, the intermediate object no longer would be useful to the execution of the program, and we would like the compiler to eliminate the object creation and the memory stores as unnecessary.

As another example, consider the standard **`to:Do:`** looping control structure. This control structure takes two integers and a block as arguments and iterates through all the integers between the two integer arguments, invoking the block for each integer. Perhaps a better way of defining a **`for`** loop would apply the standard **`do:`** control structure (which iterates over an arbitrary collection) to an interval object (which represents the collection of all integers between its lower and upper bounds). In SELF, an interval can be created by sending the **`to:`** message to an integer with an integer argument, so **`for`** loops could be written as follows:

```
(lowBound to: upperBound) do: [ "body of the loop" ].
```

Intervals are implemented as follows:

```
traits interval = ( |
  parent* = traits collection.
  ...
  do: aBlock = ( | i |
    i: lowerBound.
    [ i <= upperBound ] whileTrue: [
      aBlock value: i.
      i: i successor.
    ].
    self ).
  ...
| ).
interval = ( |
  parent* = traits interval.
  lowerBound.
  upperBound.
| ).
```

with the interval creation code defined for integers:

```
traits integer = ( |
  ...
  to: upperBound = (
    (interval clone lowerBound: self) upperBound: upperBound ).
  ...
| ).
```

This design would economize on the number of concepts needed for normal SELF programming; **`do:`** is a well-known operation in SELF, and intervals are a useful data structure in their own right. Programmers would not need special iterator methods just for integer loops.

In typical usage, this style of **`for`** loop would create a new interval object for each invocation of the loop. If performance comparable to a traditional language's **`for`** loop built-in control structure is desired, the overhead of creating this interval object must be reduced. Fortunately, the interval object is created and then almost immediately thrown away after the **`do:`** method for intervals extracts the lower and upper bounds of the iteration, with no remaining run-time uses, and so we would hope that the compiler could optimize away the object creation overhead entirely.

Unfortunately, the SELF compiler currently cannot optimize away object creations and stores. Eliminating object creates is complicated by source-level debugging. If the debugger is invoked when the object is in scope and therefore visible on a stack dump, the compiler must provide enough information for the debugger to at least create the illusion at debug-time that a real object was created and initialized. This problem could be avoided in some cases if the compiler was able to determine whether the object could ever be visible at debug-time; this analysis would be much like determining whether the created object was ever "exposed" to the outside world. The compiler could eliminate any stores into an un-exposed object (since those stores could never be seen by other routines or the SELF programmer) and subsequently eliminate the object creation code itself (since all uses of the created object would be gone).

With the current SELF coding style, this optimization is not crucial for good performance, although it would be helpful. However, if the new style of **`for`** loops using intervals and **`do:`** were to be implemented efficiently, or if other aspects of SELF programming style were to change, then this optimization would be needed to maintain the current high level of run-time performance.

9.7 Summary

The SELF compiler uses type analysis to propagate information about the types of variables and expressions through the control flow graph, to maximize the benefits received from the relatively scarce type information that the compiler can infer. The compiler maintains several data structures as part of type analysis. The mappings from names to values and from values to types are central, being used to determine the type of the receiver of a message (in support of compile-time message lookup and inlining) and the type of arguments to a primitive (in support of eliminating run-time type-checking overhead). Type testing code alters the value/type mapping, implicitly altering the induced name/type mapping for all names aliased to the tested value, as is necessary in a system relying on aggressive inlining.

Type casing exploits the information contained in union types by inserting run-time type tests that branch to monomorphic versions of code that are amenable to compile-time message lookup and inlining. Type prediction uses context information in the form of the names of the message sent to an expression to make a prediction about the likely type of the expression. This prediction is exploited by replacing the original type of the predicted expression with a union type that contains both the predicted type(s) and the original type. Type casing then is employed to read the new union type information and insert the appropriate run-time type tests that verify the prediction and branch to optimized code in the case that the prediction is correct.

The compiler optimizes blocks as part of type analysis, primarily by deferring the creation of a block until its first run-time use, if any. The compiler also calculates which blocks have been "exposed" to the outside world, and weakens the types of only those variables that are potentially up-level assigned by exposed blocks.

The name/value mapping is also used to support common subexpression elimination of redundant arithmetic calculations, since the result values of arithmetic calculations are structured and so can be compared for equivalence. An additional mapping from cells to values supports common subexpression elimination of redundant memory fetches and stores, in turn allowing type analysis to track the types of values stored into and later fetched out of assignable data slots in the heap.